TU Clausthal

Sebastian Herold

Architectural Compliance in Component-Based Systems

Foundations, Specification, and Checking of Architectural Rules

SSE-Dissertation 5

Software
Systems
Engineering

Institut für Informatik
Lehrstuhl von Prof. Dr. Andreas Rausch

Architectural Compliance in Component-Based Systems

Foundations, Specification, and Checking of Architectural Rules

D o c t o r a l T h e s i s
(D i s s e r t a t i o n)

to be awarded the degree of
Doctor rerum naturalium
(Dr. rer. nat.)

submitted by
Sebastian Herold
from Düsseldorf

approved by the Department of Informatics,
Clausthal University of Technology

2011

Dissertation Clausthal, SSE-Dissertation 5, 2011

Date of oral examination: September 5, 2011

Abstract

The intended software architecture of a software system manifests the earliest and most fundamental design decisions. To ensure that the final software product is consistent with those design decisions and the requirements realized that way, the software architecture has to be refined correctly. This means that the artefacts of detailed design and implementation have to comply with the intended software architecture.

A basic task to ensure architectural compliance is the checking of *architectural rules*. These rules are constraints resulting from the application of architectural principles, like patterns, reference architectures or guidelines, and restricting the way an architecture can be refined. However, checking those rules is difficult. Manual checks are in general not possible due to the size and complexity of modern software systems. Architectural rules are often described only implicitly and informally, partially due to missing description techniques; thus, realizing powerful tool support is challenging. Moreover, compliance checking tools have to be very flexible. The great variety of architectural rules requires expressive formalisms; in realistic scenarios, furthermore, rules have to be checked in many different artefacts, like code of different programming languages.

This thesis develops an approach to flexible architecture compliance checking in model-based development approaches for component-based systems. It describes a conceptual framework representing component-based systems as relational structures. Models are interpreted as first-order logic statements about those structures; architectural rules, which are considered being fundamental part of architectural models, are logical statements, too. Meta model-specific transformation definitions specify, how instances of the meta model are transformed into logical statements, and which architectural rules are generated if applicable. Compliance between models can be expressed this way as semantically-founded relations between logical formulae.

The developed concepts are evaluated by a case study system following several different architectural principles. Architectural rules are developed for those principles and are checked for compliance. Furthermore, the concepts are implemented by a prototypical compliance checking tool basing upon a logic-based knowledge representation and reasoning system.

Both, the conceptual framework as well as the prototypical implementation, allow very flexible architectural compliance checking. Due to the developed formalization, a broad range of architectural rules can be specified, largely meta model-independently. Integrating new meta models to check compliance between their instances and a software architecture is easy. Aside from the significant improvement of tool support compared with the state of the art, the approach fosters furthermore a new understanding of the role of software architecture; as blue-print for design and implementation, the need for explicitly modelled architectural rules is emphasized.

Acknowledgement/Danksagung

Auf dem Weg zur Fertigstellung dieses Büchleins haben mich viele Menschen begleitet. An dieser Stelle möchte ich mich bei all denjenigen bedanken, die somit bewusst oder unbewusst zum Gelingen dieser Arbeit beigetragen haben.

Ich bedanke mich bei Prof. Dr. Andreas Rausch, meinem Doktorvater, der mich immer ermutigt hat, das Thema der Architekturkonformität anzugehen und zu verfolgen. Die Diskussionen mit ihm zum Thema dieser Dissertation im Speziellen und zu Softwaretechnik im Allgemeinen habe ich immer als fruchtbar und förderlich empfunden. Des Weiteren bedanke ich mich bei ihm für die abwechslungs- und lehrreiche Arbeit, die ich als Mitglied seiner Arbeitsgruppe in den letzten Jahren ausüben durfte.

Prof. Dr. Niels Pinkwart danke ich für seine Bereitschaft, recht kurzfristig als Zweitgutachter der vorliegenden Dissertation zu fungieren.

Außerdem bedanke ich mich bei Dr. Clemens Szyperski als weiteren Nebengutachter. Trotz engen Zeitplans hat er es sich nicht nehmen lassen, wertvolle Anmerkungen zur Verbesserung dieser Arbeit zu geben.

Mein besonderer Dank gebührt zudem meinen Kollegen des Lehrstuhls für Software Systems Engineering, insbesondere den Personen, die mit mir in Projekten zusammen gearbeitet, Vorlesungen betreut oder wissenschaftlich diskutiert und publiziert haben. Fast noch mehr als die konstruktive Zusammenarbeit wird mir die Atmosphäre am Lehrstuhl in Erinnerung bleiben. Der fast unerschütterliche und sehr spezielle Humor des Lehrstuhls hat mir insbesondere in schwierigen Phasen des Schreibens sehr geholfen! Ich hoffe, dass das “Biotop SSE”, der Zusammenhalt und die Lehrstuhlküche bzw. der Kickertisch als Horte des Absurden erhalten bleiben und noch vielen von Euch auf dem Weg zur Promotion helfen [Cha01].

Ich danke Stefan Wittek und Patrick Dohrmann, die durch ihre Hiwi-Tätigkeiten und Abschlussarbeiten Teile des Prototyps bzw. wichtige Vorarbeiten realisiert haben. Für die durch penible Anmerkungen zu Euren Arbeiten und die durch DesignIt entstandenen Unannehmlichkeiten möchte ich mich entschuldigen.

Für das Review dieser Arbeit und inhaltliche Diskussionen zu Architekturthemen bedanke ich mich bei Constanze Deiters. Alle im Text noch enthaltenen Fehler und Unstimmigkeiten habe ich erst nach ihren Reviews eingefügt und konnten daher nicht von ihr entdeckt werden! Über das Korrekturlesen hinaus hat Constanze mich zudem stets moralisch unterstützt, den Rücken frei gehalten (und gestärkt) sowie meine literarischen Launen ertragen. Für den Glauben an mich, die Geduld und das Verständnis bedanke ich mich und hoffe Dir die gleiche Hilfe sein zu können.

Bei Holger Klus bedanke ich mich ebenfalls für das Review von Teilen dieser Arbeit sowie für den Erfahrungsaustausch während der Promotionszeit. Wegen des fast synchronen Beginns unserer Zeit als wissenschaftliche Mitarbeiter und unserer tatsächlich synchronen Umzüge ins schöne Goslar, waren unsere Gespräche zu Dissertation und Gott und den Harz für mich besonders wertvoll.

Meiner Familie, insbesondere meinen Eltern, danke ich für ihre Unterstützung in allen Bereichen des Lebens während meiner akademischen Ausbildung und bisherigen Laufbahn.

Meinen Freunden gilt mein tief empfundener Dank für freizeitliche Ablenkung und Zerstreuung, insbesondere aber auch dafür, dass sie mich nach langem dissertationsbedingtem Abtauchen noch als einen der ihren erkennen! Ihr habt meinerseits in letzter Zeit bei weitem nicht die Aufmerksamkeit bekommen, die ihr verdient hättet. Sofern gewünscht, wird dies nun wieder anders!

Sebastian Herold Goslar, im Juni 2011.

Contents

List of Figures

List of Tables

List of Definitions

Chapter 1.

Introduction

Contents

Architecture is music in space,
as it were a frozen music.

Friedrich Wilhelm Joseph von Schelling

It is of vital importance in most engineering disciplines that the products of design processes are compliant with the plans or models that describe the intended product. Creation, maintenance, and the operation of such products become error-prone and expensive tasks if the models are inconsistent and do not describe the actual product. This is also true for the engineering of software systems; they are highly complex and subject to frequent changes. Hence, compliance with models and plans of systems is often violated, and inconsistencies arise. This works deals with the problem of compliance with the software architecture of software systems and proposes an approach to flexible compliance checking.

1.1. Motivation

A large number of definitions and considerations for the term “software architecture” can be found in research and industrial practice [CBB$^+$10, GA02, HN99, TMD09, WR05]. Although they differ in the details, many of the definitions are similar to that used in [BCK03], which states that a software architecture of a software system is the “fundamental organization of a system, embodied in its components, their relationships to each other and the environment.” In other definitions, “components” is replaced by “building blocks” or “software elements” but at the core they are similar. Some emphasize additionally that there is no single software architecture for a system but different views like logical, distribution, or physical architectures. The differences are not that important, since the most important point in all of them is that software architecture deals with the “fundamental” structures of a system. It neither contains implementation details of single components nor a detailed design; it abstracts from details and determines those structures that influence the shape and the behaviour of a system at the most general level.

The specification of a software architecture manifests hence the earliest and most far-reaching design decisions that are made during the development of a system [BCK03]. In most software development processes, the design of an appropriate software architecture for a system (or at least of parts of it) is the first step after requirements elicitation. The subject of the engineering process switches from elaborating the “what” to build to the “how” to build the system. The selected or created software architecture influences significantly functional, non-functional, quality, and life-cycle properties of the system to be built. Of course, the goal of a specified intended software architecture is to guarantee properties for the system that fulfil the identified requirements at a highest possible degree [GS94, GP95].

Due to the fundamental character of the software architecture and the design decisions it embodies, it is hard and costly to repair or to change afterwards during refining design, implementation, and maintenance [Bal00, Som10]. Changing a software system at the architectural level, because, for example, the existing software architecture was inappropriate or could not fulfil emerging requirements, means to change a system in its most basic structures. Even if it is possible at all, the changes will most probably result in massive restructuring efforts in design specifications and code.

But even if we are sure, or even if we validate, that an appropriate software architecture for a system has been specified, the intended software architecture will not be sufficient to guarantee a system with correct functionality and the required quality attributes. Inadequate detailed design and implementations can always undermine an appropriate software architecture and lead to systems whose software architecture violates or contradicts the intended software architecture.

It is very likely that, during the development of modern software systems and applications, the actual software architecture will diverge from the intended one. This effect is often called “architectural erosion” [PW92]. With increasing size and complexity of software systems, and their degree of distribution and their longevity, the probability of architecture violations increases, too. The complexity of the processes to develop, to maintain, and to run such large and complex systems adds to the danger of architectural mismatches.

Even highly sophisticated experts like the Eclipse developer team are not able to stick completely with their software architecture [BBS03]. Simple rules regarding import dependencies between packages are violated — hence the actual architecture differs from the intended one with the possible consequences described above.

Moreover, modern software systems and applications are often part of even larger conglomerates of systems. Enterprise applications, for example, are embedded in enterprise-wide IT-landscapes [JE07]. Future software systems will be more and more integrated into so called systems-of-systems or IT-ecosystems [Nor06, MS03, HKNR08]. These overlying structures with their own architectures may additionally constrain the design of single applications by specifying application-spanning reference architectures, patterns and so on. This makes it even harder for software architects and designers of software systems to guarantee the compliance of the system with all of its architectural aspects.

Some definitions of software architecture pick up the issue of valid refinements and state in similar ways like [BCK03] that a software architecture “contains the principles governing design and evolution.” In other words, the software architecture of a system provides a framework for the system’s detailed design and its implementation. Consequently, it has to be ensured by software architects, software designers, and programmers that a software architecture is refined correctly by detailed design and the implementing code. Only in this case, we can be sure that the functionality and the quality attributes guaranteed by the specified software architecture find their way into the implemented system.

One has to come to the conclusion that it is essential for the quality of a software system and its successful creation, maintenance, and operation that the detailed design and the implementation of a system is regularly checked for their compliance with the intended software architecture. It it obvious that this cannot be done by hand due to the size and complexity of modern software systems; proper tool support is required. There exist many tools for this task. The tool SonarJ, for example, is able to check whether import statements in Java source code are compliant with a logical layering of the system given by the software architect.

Unfortunately, current tool support is not flexible enough to easily check architectural compliance for complex systems in all its facets. First, most tools address only a single or few architectural aspects. As already mentioned, SonarJ, for example, focuses on architectural layers. Especially in large systems, the software architecture is constructed by many different relevant aspects connected like building blocks and forming the overall software architecture. A diverse set of architectural patterns or styles, reference architectures, guidelines, and policies is used. Each of these building blocks may define its own independent constraints for the refining design and implementation. To our best knowledge, there is no single tool covering a broader range of possible compliance checks towards complex architectures, or a tool that could easily be extended to do so. Second, most tools do not consider the different kinds of artefacts whose contents could probably be affected by architectural constraints. In complex systems, it is likely that different modelling languages are used for different parts of the systems, that more than one programming language is used for implementation, and that additional artefacts like configuration files contain information that should be included in the compliance checks. Concluding from these two issues, a hard to handle and probably expensive tool chain would be necessary for the task of architectural compliance checking.

However, beneath the problems of current tool support, there is a deeper and more principal problem that has to do with the way software architectures are described and understood. Often software architectures are not described or specified explicitly, or without formal syntax or semantics as a kind of ‘whiteboard architecture’. Compliance checking supported by tools is simply impossible in this case. But more interesting, even description techniques and approaches with formal syntax and semantics that claim to describe systems at the architectural level do not solve the problem. None of them provides description techniques at a level of abstraction allowing us to tell how the specified ‘architecture’ is further refined by detailed design — instead the description contains a detailed design itself. For example, most component-based approaches focuses on the definition of components and interfaces with their methods. This is a much more detailed view onto the system than describing which layers it consists of, and what the layering means to the way components may be structured or may communicate. Hence, there are no seamless approaches to describe component-based software systems from software architecture down to software design and implementation. While component-based approaches suit as an abstraction from implementation, there remains a gap that need to be bridged between those approaches and software architecture development.

In conclusion, the main problem lays at the heart of architecture description. We need to define the "architectural rules" for a software architecture and its elements. These rules define how the software architecture can be refined during design and implementation. For this purpose, an adequate description technique for software architectures is required. Without addressing these issues, architectural rules cannot be properly defined and tool support cannot be improved.

1.2. Goals and Contribution of this Work

The goal of this work is to provide an effective approach supporting software architects, designers, and programmers to develop and maintain architecturally compliant component-based software systems. For this purpose, it aims at facilitating a formal basis for flexible architecture compliance checking tool support. This work introduces a logic-based conceptual framework for architectural compliance checking to tackle the above mentioned challenges that hinder efficient tool support so far. The framework consists basically of three main components:

- A formal representation of component-based software systems as relational structures and a formalization of descriptions of such systems, like architectural models, design models or implementing code, as first-order logic statements over such structures. The formal representation allows one to abstract from specific component-based description techniques and implementation languages. Components are understood as extension of classes as concept of object-orientation; components are complex, instantiable types. The functionality that a component provides or requires is specified by interfaces defining method signatures. The provided interfaces of a component are implemented by component-local classes. Instances of components consist of a network of linked objects at runtime. The structure of this network is specified by parts. Parts describe the objects and the links between objects of the instantiated network, and specify those objects that can be accessed from the environment of a component instance (provided ports), or that can access objects in the environment (required ports). Hierarchical components can be constructed by typing parts with components. Communication between components takes place as synchronous or asynchronous invocation of methods between ports.

- A formal classification criterion based on the formal representation of system descriptions to distinguish clearly and precisely between architectural, detailed design, and implementation descriptions of component-based systems. The classification criterion defines categories of first-order logic statements. These categories can be mapped to the statements that should be made during the different phases of designing a software system, i.e. architectural design, detailed design, and implementation. Based upon this, those constraints will be defined that have to hold for architecturally compliant design descriptions.

- An operationalisation of the concepts mentioned above for the model-based development of component-based software systems. This includes particularly how to derive the formal representation of models as logical statements, including the architectural rules defined by architectural models, and how to perform the compliance checks efficiently.

The developed concepts are evaluated by applying them to a practically relevant case study. It describes a realistic component-based software system with different architectural aspects. For these aspects, architectural rules will be defined according to the proposed conceptual framework. The case study system will be checked for architectural compliance regarding these aspects in different scenarios.

Furthermore, a prototypical implementation of an architectural compliance checking tool is provided. It is able to interpret the architectural rules defined by a description written in the developed architecture description language and to check them in UML-based component-based design specifications and Java source code.

1.3. Structure and Content

This thesis is structured as follows. In Chapt. 2, *Software Architecture Compliance - A Problem Analysis*, the problem of checking architectural compliance in today's software development will be analysed. It will start with the introduction of some general terms and a discussion of the roles of architectural design, detailed design, and implementation in modern software development, and the general difficulty to ensure architectural compliance. Afterwards, the problem analysis will take a more focused look upon a single class of systems and one specific development approach — component-based systems developed by following model-based approaches.

Related approaches will be subject of Chapt. 3, *State of the Art*. Different classes of approaches to architectural compliance checking will be discussed as well as methods and principles of related fields. It will be examined why existing approaches to compliance checking are not sufficient to provide flexible and adaptable tool support.

Chapter 4, *Case Study*, will introduce the case study illustrating the need for flexible architecture checking, and which will serve as application scenario for the proposed approach. Case study is the *Common Component Modelling Example (CoCoME)* which is an artificial but realistic, w.r.t. practical relevant size and complexity, component-based system. A reference design model will be checked for architectural compliance as well as modified variants of the system reflecting realistic modification scenarios. In these modification scenarios, different architectural aspects of the system will be violated; these violations need to be detected by architectural compliance checks.

Chapter 5, *A Formal Framework for Architectural Compliance Checking*, will describe the proposed solution to flexible architecture compliance checking in depth. After introducing the core ideas and some mathematical foundations, the formalization of component-based systems themselves and descriptions of them will be explained in detail. Based on this formalization, formal definitions of architectural rules and architectural compliance can be specified. To be able to provide tool support, an operationalisation of the formal concepts will be introduced.

In Chapt. 6, *Architectural Rules for UML as Architecture Description Language*, the proposed framework will be utilized to define architectural rules for UML. For this purpose, a UML profile will be introduced that covers the architectural aspects relevant for the case study. Hence, the software architecture of the CoCoME system can be described as UML model to which that profile is applied. It will be specified how architectural UML models are transformed into the formal representation required for architectural compliance checking. Moreover, the architectural rules are defined for the considered architectural aspects in the case study.

The results of executing the actual architectural compliance check of the CoCoME case study are discussed in Chapt. 7, *Architecture Compliance Checking of the Case Study*. First, the reference design model will be checked against an architectural model of the system given in the tailored UML dialect defined before. Then, the checks in the different modification scenarios introduced in Chapt. 4, will be presented as well as their results.

Chapter 8, *Design of a Logic-Based Compliance Checking Prototype*, will present a prototypical implementation of the proposed concepts. The architecture and design of the prototype will be introduced. It is able to execute architectural compliance checks based upon the knowledge representation and reasoning system *PowerLoom*. Models participating in a compliance check are represented as logical knowledge base upon which the architectural rules are executed as logical queries. The prototype is based upon a framework allowing us the easy integration of models, source code, or other artefacts of different languages or schemas, and to check their architectural compliance. This framework can be easily reused as backend for arbitrary different architecture checking tools following the conceptual approach of this work.

The final Chapt. 9, *Conclusion*, will discuss the results of this thesis. The contributions of the thesis regarding the tackled research questions will be summarized and the limitations of the proposed approach will be examined.

Chapter 2.

Software Architecture Compliance - A Problem Analysis

Contents

Always design a thing by considering it in its next larger context — a chair in a room, a room in a house, a house in an environment, an environment in a city plan.

Eliel Saarinen

The main title of this work is “Architectural Compliance in Component-Based Systems”. To motivate the presented research and understand the problem that has lead to the proposed approach, it is necessary to have a common understanding of the underlying domains or fields. The single pieces of the main title suggest that the relevant fields are *architecture*, in the following always understood as *software architecture*, *compliance* or *conformance* management, and *component-based* systems.

This chapter will analyse the problems in detail leading to the need for a flexible approach to architectural compliance checking. Section 2.1 will give a quick overview on foundations of software architecture and the tasks of software architects. Building on that, Sec. 2.2

will especially address the relationship between software architecture, detailed design and implementation of software systems; it addresses the "compliance piece" of the title. It will analyse the problems regarding conformance between software architecture and detailed design in general, and will deepen that analysis for the special case of model-driven software development (MDSD) approaches. Foundations of component-based systems, as final element of the overall title and topic, will be covered in Sec. 2.3.

Section 2.4 will conclude the basic research questions derived from the problem analysis. These research questions guide the further work; Sec. 2.5 sums up this chapter.

2.1. Foundations of Software Architecture

The roots of the field that is called *software architecture* today date back to the early 1970's when the just founded, or at least explicitly named software engineering community [NR69] started to deal with the fundamental principles of system design. Researchers like Edsger Dijkstra and David Parnas, among others came up with concepts like information hiding, layering of systems, views, and many more [Dij68, Par72, Par74]; terms that are relevant for system design in general and for architecture specifically. Nevertheless, there is still no common and broadly accepted definition of the term *software architecture*.

This section introduces some foundations of software architecture, especially how the term is used in the remaining book. Furthermore, it will give an overview of the tasks a software architect usually faces in practice; especially the task of setting the rules for design and implementation and checking them will be illustrated.

2.1.1. What is Software Architecture?

There exist a large number of more or less adequate definitions of the term *software architecture*. The website of the renowned Software Engineering Institute of Carnegie Mellon University [Sof] lists about 150 definitions given by members of the software architecture community. A common core that is part of many of the substantially valuable definitions states the main subject: software architecture is about software elements (or components, building blocks, etc.) in a software system and the relationships between them.

However, this is a very general essence of many definitions. One of the definitions containing this essence is that one given in the IEEE Standard 1471–2000 "Recommended Practice for Architectural Description of Software-Intensive Systems" [IEE00]. It defines the software architecture of a system as:

> "The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution."

This definition adds two properties to the mentioned common core. First, software architecture is not about every detail in the structure of a software system but about the "fundamental organization". The name of a loop variable in a function, for example, is surely not of architectural importance. Second, the definition emphasizes that software architecture is not about components alone but also about principles for the development of the system. For example, the layering of a system [BMRS96] defined by its architecture, has implication on its implementation; there might be system parts, for example classes or functions in the lowest layer, that cannot call methods at classes from layers above. This means that a software architecture restricts the way a software system can evolve and be refined.

In addition to the definition above, [BCK03] adds some important aspects and states:

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

First of all it adds that there is no single architecture of a system but potentially many; this reflects the broadly accepted fact that many different views on a system make up the overall software architecture; see, for example the 4+1 architectural model [Kru95] or the architectural ontology defined by the already mentioned IEEE Standard 1471-2000 [IEE00]. This especially includes behavioural views on the software systems; the terms "relationships" and "component"/"elements" in both definitions suppose that only static structures are of interest in software architectures. This is clearly not the case. The second addition of that definition gives also a hint for a differentiation between software architecture and further fine-grained design. Software architecture is about components and their visible properties — i. e. basically their interfaces in the broadest sense. The internal details of such components are subject to software design and implementation. The definition, however, does not say at which granularity an element in a system should be considered to be a component; at both extremes, subsystems and methods could be seen as components in the architectural sense, with huge implications on what is subject to architecture development and to fine-grained design.

The informal definition of the term "software architecture" that will be used in the following concludes the definitions above:

> "The software architecture of a software system is the structure or structures of the system, which consists of components, their externally visible interfaces, and the relationships among them. Furthermore, it defines *architectural rules* that restrict and guide the further design and implementation of the architecture, as well as its evolution."

The sources for architectural rules are diverse:

Patterns. Architectural patterns like *Layers* or *Pipes-and-Filters* [BMRS96] define constraints and guidelines how components and interfaces have to be realized, and how they should interact.

Reference Architectures. These are renowned architectural, often domain-specific blueprints of how to build systems, for example the 3-Tier-Architecture for distributed enterprise systems [TvS08]. They influence the implementation of a system as well.

Design Principles. They describe basic guidelines for the design of systems or properties that should be ensured to be able to create high quality software. For example, the *Law of Demeter* [LH89] or the *Separation of business and technical concerns* [Eva03] are principles, which of course, imply constraints on design and implementation.

It has also to be mentioned, that rules originating from the same source may differ from application context to application context. Often there exists alternative realizations of patterns, or there are project-specific exceptions from design principles. Hence, architectural rules are neither globally defined in general nor carved in stone but might be customized for specific purposes.

2.1.2. Tasks of a Software Architect

The term “Software Architect” is rather the name of a certain role that people can play in software development project than a specific profession based upon a dedicated education. Understood as a role, a software architect is someone who is responsible for the software architecture of a system.

Among the major tasks of software architects are the creation and maintenance of software architectures. A software architect has to create an appropriate software architecture for a software system; “appropriate” means that the architecture has to fulfil the given functional and extra-functional requirements as far as possible. For this purpose, the software architect has to clarify requirements and general conditions, and must resolve conflicts that arise from competing requirements. He has to decompose the system and to identify components and interfaces. Furthermore, he has to deal with cross-cutting technical issues that affect the system during design, testing, and operation; for example, he has to evaluate operating systems, programming languages, technological frameworks, etc., if necessary. Maintaining an existing software architecture means to correct architectural errors, adapting the architecture to changing requirements, and avoiding that intended and actual architecture diverge.

In order to create and maintain software architectures, the architect has to cooperate with many stakeholders during diverse subtasks. Figure 2.1 shows some of those stakeholders influencing the work of a software architect and interacting with him. The arrows are annotated with examples of tasks he performs for, or together with the connected group of stakeholders, or responsibilities he has to fulfil with regard to certain stakeholders. For example, he is the prime contact for the project management regarding technical issues of a software development project and the design decisions that are made. Together with the project manager, he plans how the development team is staffed; this is massively influenced by the architecture because the decomposition of the system affects the structure of the development team and its ability to develop system parts in parallel. Figure 2.1 shows that many different stakeholders have to be considered to clarify the requirements that a software system, and hence its architecture, has to fulfil.

The highlighted relationship in Fig. 2.1 between software architect and software designers / software developers is of special interest for the further investigations. One of the responsibilities of the architect regarding the design and development is to control that the architecture is correctly refined and implemented, i.e. that the design and implementation are architecturally compliant. On the one hand, the specified components, interfaces and relationships specified by the structures of a software architecture have to be correctly reflected and implemented; on the other hand, the architectural rules have to be followed.

This task is very important with regard to the overall quality of the software system, as discussed in Chapt. 1. All the valuable work of requirements engineering and software

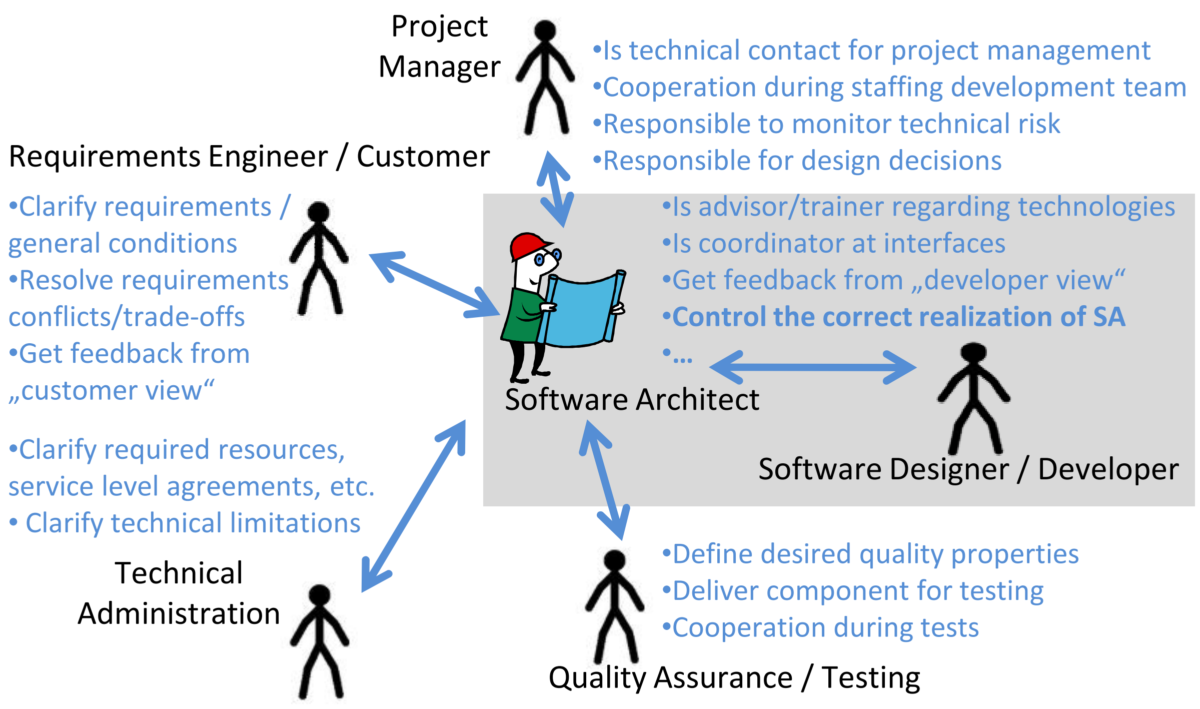

Project Manager
•Is technical contact for project management
•Cooperation during staffing development team
•Responsible to monitor technical risk
•Responsible for design decisions
Requirements Engineer / Customer
•Clarify requirements / general conditions
•Resolve requirements conflicts/trade-offs
•Get feedback from „customer view“
•Is advisor/trainer regarding technologies
•Is coordinator at interfaces
•Get feedback from „developer view“
•Control the correct realization of SA
•...
Software Architect
Software Designer / Developer
•Clarify required resources, service level agreements, etc.
• Clarify technical limitations
Technical Administration
•Define desired quality properties
•Deliver component for testing
•Cooperation during tests
Quality Assurance / Testing

Figure 2.1: Tasks and responsibilities of software architects and interacting stakeholders.

architecture design can be eliminated by incorrect refinement and implementation. If the actual software architecture does not conform to the intended architecture, the actual system is likely not to have the quality attributes expected by the stakeholders.

It is obvious that the task of compliance checking cannot be performed manually. Current large-scale systems have up to several hundred million lines of code; large object-oriented systems are designed of several thousands of classes. Ensuring architectural compliance manually for systems of that size and complexity is impossible, and even for smaller systems time-consuming and error-prone. The software architect clearly requires tool support.

2.2. Compliant Realization of Software Architectures

In this section, we will investigate the factors causing the definition of architectural rules, checking them, and ensuring their validity to be tough tasks. First, some general observations will be made; the second subsection will deal with specific issues of model-driven development approaches.

2.2.1. Software Architecture, Detailed Design, and Implementation

After conducting the requirements of a software system, the inner structures of a system are developed. While requirements engineering elaborates what has to be build, the design of the inner structures aims at how to build it. Note, that although the description might suggest

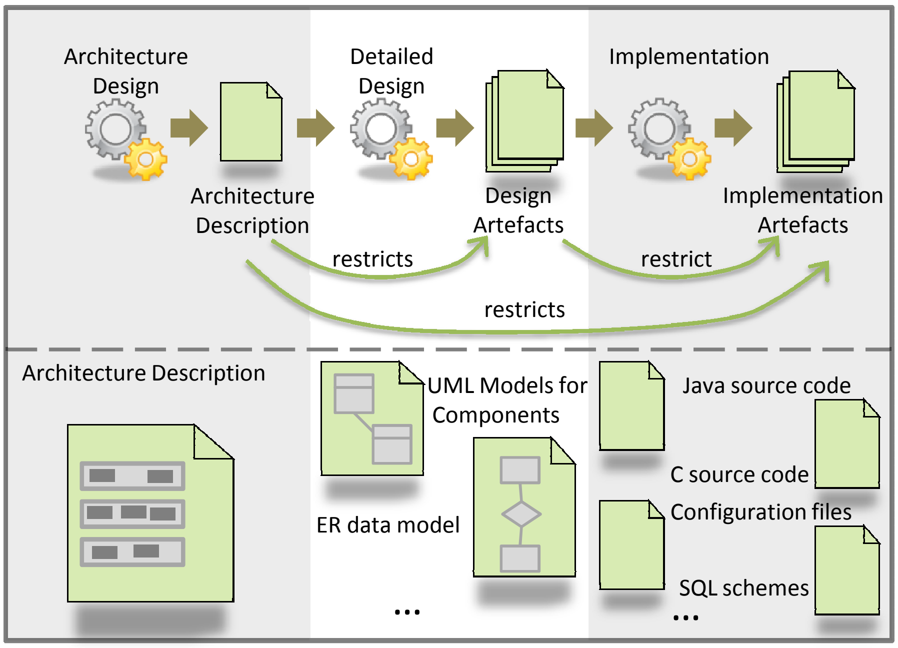


Figure 2.2: The three levels of software development; the lower part shows examples of artefacts created during development of an information system.

so, both are not strictly chronologically separate phases; depending on the *process model* or *strategy*, like waterfall model, incremental or iterative development [Bal00, Som10], both tasks are executed partially, in parallel, or repeatedly.

In the explanations so far it was implicitly assumed that the overall design process consists of three different steps, namely architectural design, detailed design, and implementation, which, again, are not necessarily separate phases. Artefacts created in those steps provide different views on the inner structures of a system with different levels of abstraction, adding more and more details starting at the most abstract view of the software architecture. As defined in Sec. 2.1, the software architecture contains components and their externally visible properties, their interfaces. These structures are refined during the detailed design and complemented by the inner structures of components. The implementation, finally, provides the complete and executable system, or, at least, artefacts that are automatically transformed into executable elements, for example, source code transformed into executable binaries by compilers.

Figure 2.2 illustrates this rather short and simplified explanation of the overall design process. At each level before implementation, developed artefacts restrict the further refinement. For the architectural design level, these constraints are exactly those architectural rules. The lower part of Fig. 2.2 shows a simple example. Let us assume that the system under consideration, is an industrial information system to manage the product portfolio of an enterprise and to manage customer data. The software architect designs a component in the architecture, which is responsible for saving the product portfolio in a persistent database. Moreover, he states that the component must be independent from any other component in the architecture — clearly an architectural restriction for the refinement. The software designer responsible for the detailed design adds details to this component; for example, he creates an Entity-Relationship-Model

[EN10] of the data model which is the base for the functionality of the component, like creating, modifying, saving, and removing products from the portfolio. Moreover, he creates design models written in UML [Obj10b] to describe the classes realizing database access, queries, and so on. These models have to conform to the architectural rule prohibiting dependencies towards other components; it has implications on the embedding of the component's data model in the overall data model of the system; moreover, the UML design model cannot introduce dependencies to other components.

Later, the developer takes this information, for example, to create a Java implementation of those classes and to create database schemes from the ER-model. Further artefacts might be created or integrated, like existing legacy libraries written in C. All of these artefacts can directly or indirectly be affected by architectural rules.

Inherent Complexity of Compliance Checking

Assuming that architectural compliance cannot be guaranteed from scratch, i.e. it cannot be ensured a-priori that design and implementation are compliant, checking compliance is required. This is a difficult task because of the inherent complexity. This complexity is due to different factors.

First, the heterogeneous set of artefacts that have to be checked, adds to the complexity of the problem. As the example above shows, there are many types of artefacts that result from single steps of the overall design process, e.g. UML-based design models, ER models, source code of possibly different programming languages, and technology-specific artefacts. All of those types of products can violate architectural rules and have to be checked.

The potential diversity of architectural rules is the second factor of inherent complexity. The sources of rules are manifold; patterns, reference architectures, and design principles can cause constraints for the further refinement of architectures. In each of these groups, the aspects the elements address can vary widely. For example, while the layers pattern addresses the logical structure of a systems, there are patterns affecting primarily distribution aspects or behavioural aspects. Furthermore, the set of rules is not fix in general but can contain user specific rules that, for example, exist due to enterprise-specific design guidelines, or specific forms of patterns and reference architectures.

It is desirable that tool support for architectural compliance checking addresses as many aspects of architectural rules as possible. With this goal in mind, and considering the potential user specific tailoring of rules, the set of rules should not be fix and "hard-coded" in tool support. This means that there has to be a description technique to describe software architectures including architectural rules with the properties that

- architectural rules of a variety as broad as possible can be formulated, and
- architectural rules are described formally, thus they can be interpreted and checked by tools.

In this case, tool support can address the diversity of architectural at the best possible degree.

The Imprecise Separation of Design Levels

The inherent complexity of architecture compliance checking is tightened by the observation that the separation between the steps in the design process — architectural design, detailed design, and implementation — is most often unclear in practice and research[1]. The reason is the degree of freedom most definitions leave to the software architect regarding the granularity of software architectures. As already mentioned, the granularity of components is unclear — a subsystem can be a component as well as a single class of an object-oriented system. It is obvious that selecting different granularities affects the subjects of design at the different levels of the design process and blurs the borders between the levels. Moreover, the general definition stating that the structures given by software architectures are "fundamental" do not state precisely what "fundamental" means and which structures are considered as "non-fundamental" being subject to detailed design and implementation.

Informal attempts to define a clear separation of concerns between architectural design and detailed design are made in software architecture literature with moderate contribution to clarify things. Many definitions state in a similar way that "Architecture [...] is design at a higher level of abstraction" [Kaz99] without explaining if there is "threshold abstraction level" that separates both steps — and if there is one, how it is defined. In [CBB$^+$10], the definition of software architecture given there is used to define a separation: if software architecture is about the externally visible properties of components, detailed design is about the internal hidden properties. This definition does not help to select an appropriate granularity for software architecture and basically says: architectural are those things that the software architect defines to be architectural; everything else is subject to detailed design. Other definitions state that architecture deals with issues beyond "the algorithms and data structures of computation" [GS94] which, however, also holds for many design approaches.

As a result, a babylonian confusion of tongues can be observed regarding the terms "software architecture" and "software design" without a common understanding of the terms but with a sometimes synonymous usage instead. There are many examples in research and practice which show that the same concepts are considered being "architectural" or "design" from different points of view, or in which no clear distinction exists.

Patterns. Some patterns are sometimes considered being architectural patterns, sometimes design patterns, depending on the understanding of the particular pattern catalogue. For example, the Model-View-Controller pattern is listed as architectural pattern by [BMRS96] but as design pattern by [GHJV95].

Architectural Frameworks. Existing frameworks to plan, design, and implement architectures of systems or even system landscapes do often not distinguish between architecture and detailed design. This is the case, for example, for the *NATO Architectural Framework (NAF)*, the *The Open Group Architecture Framework (TOGAF)*, and the *Zachman Framework* [NAF07, Gro09, OFS03].

[1]The following explanations focus on the separation of architectural design and detailed design; in general, the confusion about this separation is much greater than that about the separation of detailed design and implementation.

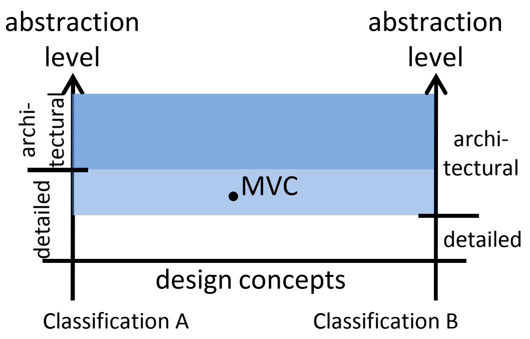


Figure 2.3: Different classifications of abstraction levels: have rules defined by the MVC pattern to be considered architectural?

Process Models. In process models like the V-Model XT [BMI08, FHKS09, RB11], there exists the principle of a system decomposition but the V-Model XT, for example, does not distinguish between the roles of a software architect and software designer; a separation of the two design steps is not made. In fact, only the role of a software architect exists in the standard. The *Rational Unified Process* [Kru03], on the contrary, uses only the term *design* and does not distinguish between architectural and detailed design.

(Architectural) Specification Languages. Languages for both levels of abstraction use the same concepts. For example, UML [Obj10b], intended as object-oriented *design* language, is also used as *architecture description language (ADL)*. Since version 2.0, it contains concepts like components, connectors, etc. to describe systems at the architectural level. These concepts were introduced initially in dedicated ADL like Darwin, ACME, etc. (see [MT00] for a list of ADL). The usage of the same language, e.g. UML, for both levels without clear guideline what has to be described in which design step, adds to the unclear separation.

But why are the impacts of unclear separation, or classification, and terminology important for architectural compliance checking? The separation of architectural design and detailed design does not only describe which structures are considered to be architectural but influences also indirectly what is considered to be an architectural rule. Hence, several understandings of architectural concepts can imply different set of architectural rules. This can have implications on the language to describe architectural rules, the required expressiveness, and the possibility to check rules automatically.

Figure 2.3 illustrates this circumstance graphically. The area between the vertical axes is the space of all design concepts and potential sources for rules guiding the further refinement and/or design of a software system, like patterns, design principles, etc. The vertical axes denote the abstraction level and are separated into the categories of "architectural" and "detailed"; each of these axes sets this classification of levels differently. Consequently, both separations cause different sets of possible *architectural* rules.

Let us now, for example, assume that the MVC-Pattern is such a source for (architectural) rules, which is considered architectural only in the right-hand classification. Furthermore, let us hypothetically say that all rules defined as architectural in both separations can be specified by a regular language, and that MVC defines rules which can only be specified by context-sensitive languages. This means that the formalism to describe architectural rules can be different, and the checking of rules may be differently complex.

Concluding, to focus on the inherent complexity of architectural compliance checking, it is necessary to define a precise classification criterion for the three abstraction classes of software development steps.

The work of Eden, Hirshfeld, and Kazman [EHK06] provides such an abstraction classification. It will be explained in more detail in Sec. 5.1 and further formalized in Sec. 5.2. In that approach, general design concepts such as patterns and artefacts describing specific systems like software architectures are understood as logical statements about systems that fulfil the specification given by a description. Comparable to the idea of architectural rules, statements can be *intensional*, i.e. they define constraints about the structures of a system in addition to the structures themselves. As distinction between architectural and detailed design concepts (or artefacts), there exist two categories of such constraints; *local* statements, once valid in a system, remain valid no matter how the system evolves or how it is refined — their impact is local in the sense, that they affect a dedicated part of the system; *non-local* statements have global impact in the sense that the impact cannot be reduced to affect certain elements only — they describe a global property of the system that can be lost at any time by evolving or refining the system.

This section has illustrated the problems in the context of architectural compliance checking abstracted from specific development approaches. In the following, we will take a look upon model-driven software development as a specific approach, which is often claimed to solve many of the consistency and conformance problems of software development, and which hence might also provide solutions for the problems of architectural compliance checking.

2.2.2. Model-Driven Software Development

The motivation of model-driven software development comes from the observation that traditional development approaches are very *code-centric*. This means that the executable code of a software system is considered as the main product of the process. Other higher-level specifications of the system, like requirements specifications or design specifications, are neglected in the sense that they are often notated as box-and-lines diagrams without precise syntax and semantics, often only as "paper" diagrams without the possibility to edit or manage them by CASE[2] tools. Some of the implications are:

- The danger of inconsistencies exists. Interrelationships between elements in different artefacts have to be inspected manually in case of changed content to propagate changes to related artefacts. Since those interrelationships are complex, this task is time-consuming and error-prone.

- Higher-level descriptions cannot be used to apply code generation for repeatedly executed task, e.g. the creation of code skeletons. Similar and re-appearing code fragments are

[2]abbr.: **C**omputer-**A**ided **S**oftware **E**ngineering

manually produced instead of being able to generate code automatically with higher productivity.

- Portability is difficult since code has to be ported manually; higher-level specifications exist only in forms that cannot be interpreted by tools to generate code of new languages or for new technologies.

These factors (and others) reduce productivity in classical development approaches and limit the ability to produce software repeatable at an high quality level. A more detailed list of problems can be found in introducing chapters of MDSD literature (e.g., [BGB10, SVC06]).

The main idea of MDSD is to shift efforts in software development from producing source code to creating implementation- and technology-independent, general and conceptual models of software systems. These models are input to model transformations generating more specific models and, at the end, the executable code. In the following, the core concepts will be illustrated by the example of the *Model-Driven Architecture (MDA)*, the framework for model-driven development proposed by the *Object Management Group (OMG)*.

General Overview of MDA

The MDA [KBW03] provides a framework in the sense that it defines a terminology and a process for model-driven development. It refers to a number of standards defined by the OMG to make the general idea of MDSD more concrete.

A *model* in the terminology of the MDA is a description of a system written in a well-defined language with defined syntax and semantics, such that tools can interpret them[3]. Syntax and semantics can be defined by a *meta model* describing the allowed structures of well-formed models, comparable to the formal grammar of a textual programming language. Another standard of the OMG, the *Meta Object Facility (MOF)*, specifies the relationship between models and meta models as well as further levels of meta modelling [Obj06].

The MDA defines different categories of models that distinguish themselves from each other by the level of abstraction of the view they provide onto a system. So-called *Computational Independent Models (CIM)* model the requirements of a system and show “the system in the environment in which it will operate, and thus it helps in presenting exactly what the system is expected to do.” [Obj03].

Platform Independent Models (PIM) describe a systems abstracting from the usage of a certain *platform*; a platform can be any set of technologies like programming languages, technical frameworks, subsystems, etc. The sense of a PIM of a system is to provide a description of the system abstracting from the actual technologies, and which hence does not change if the platform changes. This supports modifiability and portability of systems.

Platform Specific Models (PSM), in contrast to PIM, contain in their description of systems information about the usage of platforms. As a simple example, consider the UML class diagrams in Fig. 2.4. The left one represents a PIM which abstracts from a specific object-oriented programming language by depicting an object-oriented design by the common means

[3]This definition leaves some freedom to the degree the semantics must be defined, since many tools, depending on their purpose, do not require a formal semantics. Indeed, many of the languages frequently used in the MDA context, e.g. UML, do not have a precise semantics.

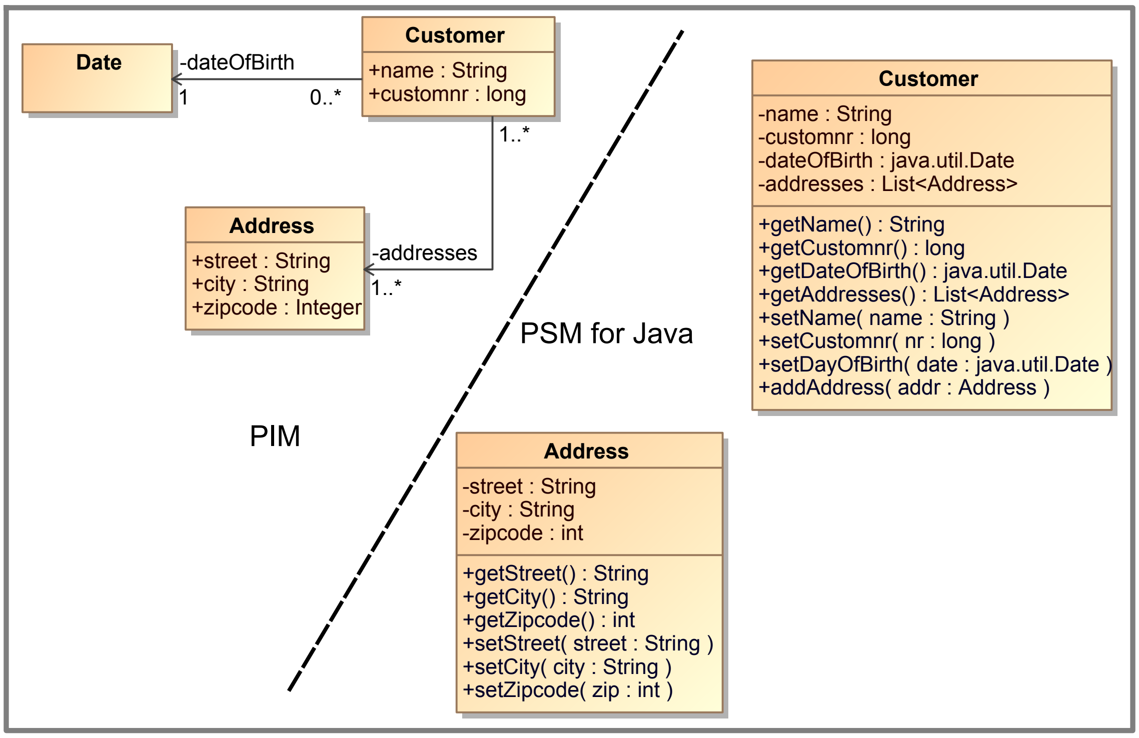


Figure 2.4: Examples of a platform-independent model (PIM) and a platform-specific model (PSM) for Java.

of class diagrams. The diagram on the right represents a PSM for Java, in which datatypes like Date are mapped to Java library classes, association ends with multiplicities greater than one are replaced by specific implementations by lists, and public attributes are replaced by corresponding private attributes with getter and setter methods.

Obviously, there is a refinement of the system descriptions from CIM to PIM, from PIM to PSM, and finally from PSM into executable code. To actually address the problematic issues of traditional approaches, MDA does not propose to do the refining steps manually but to use semi-automatic, tool supported model transformations. Model transformation is the process of converting a given source model into a target model. The overall concept is illustrated in Fig. 2.5.

Model transformations in MDA rely on executable definitions of how to transform a source model (e.g., M_1) into a target model (e.g., M_2). Those *transformation definitions* specify how elements of a source meta model MM_1 are transformed into elements of the target meta model MM_2. Transformation tools interpret these transformation definitions and execute them for the actual source model instance resulting in a generated target model.

In the example above, to transform a PIM into a Java PSM, a transformation definition had to be specified describing how UML models are transformed into UML models referring to Java specific classes and concepts[4]. The transformation definition could state that each class is transformed into a class with the same name, that attributes of a class in the source model are

[4]In this case, the meta model for source and target models would be the same, namely that of UML.

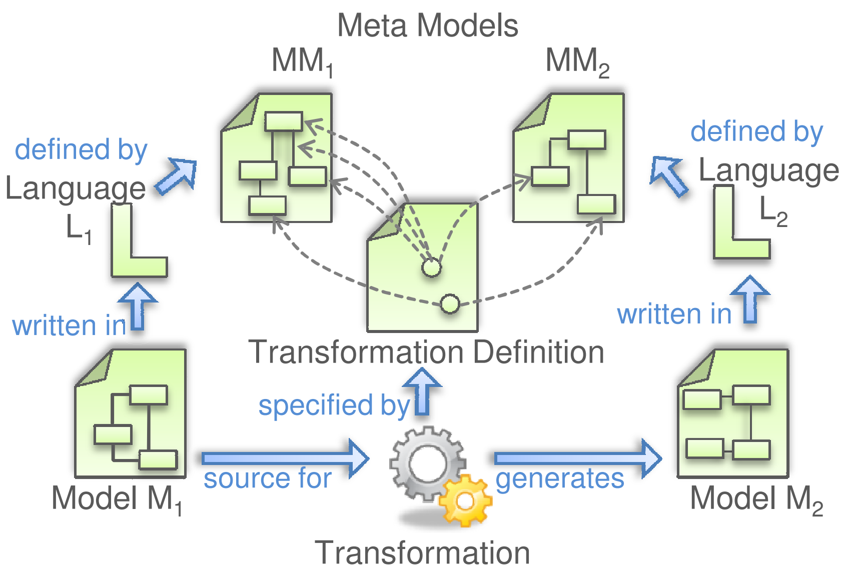


Figure 2.5: General concept of transforming a source model M_1 into a target model M_2.

made private in the Java model and complemented by getter and setter methods, and so on.

The OMG published the *Query/View/Transformation (QVT) Specification*, which is a specification for the implementation of transformation definition languages and for model transformation tools [Obj11]. The specification defines some important features of implementations, such as incremental transformations and persisting of tracing information. The first feature ensures that a repeated transformation between two models transforms only modified content and does not overwrite manual changes in the model; the second property allows tracing back the source of generated target model content after the actual transformation process; this way, for example, the class in a PIM could be determined that lead to the creation of a certain class in a Java PSM. Both features allow us, in combination with models written in well-defined languages, to address many of the mentioned problems of traditional software development, for instance, the problem of maintenance and productivity issues caused by manual consistency management.

Today, there exist a reasonably large number of QVT implementations [LS06, Med, ATL] and other approaches. Hence, conformance and consistency between models seem to be a problem easy to solve for MDSD in general and MDA in particular. In the following, we will see by showing an example written in a language implementing QVT, that there remain open issues regarding the application of MDA to architectural compliance checking.

Architecture Model Transformation Example

The example specifies a simple transformation of architectural layers into a design model using the popular *Atlas Transformation Language (ATL)* [ATL]. Since the example is very simple and straightforward, the syntax will only be explained superficially.

Imagine, we have a meta model for the definition of architectural design models, called Arch. The language defined by Arch allows us to draw boxes in our models representing layers. The corresponding meta model element is hence accordingly called Layer. As already mentioned,

Listing 2.1: ATL rule to convert layers into UML packages.

```
1  module Arch2Design
2  create OUT : UML from IN : Arch
3  rule Layer2Package {
4    from
5      l : Arch!Layer
6    to
7      p : UML!Package (
8        name <- l.name
9      )
10 }
```

layers define some architectural rules that state an element in a layer may only depend on elements in layers below their own layer. This has to hold in the detailed design as well as in the implementing code.

Now let us assume, that we do a detailed design using UML. Layers are represented as packages because there does not exist a dedicated meta model element for layers in the UML specification, and packages seem to represent them best instead — similar to layers, they can contain other elements.

The first line of Listing 2.1 indicates that the following transformation rule is part of a module of rules (which, for example, defines the transformation of other Arch elements). The second line specifies that for an instance of the Arch meta model, a UML model will be created. The specification of the actual rule starts in line three; for the element l, which is a layer (line five), a package p is created in the target model (line seven); this package has furthermore the property, that its name attribute is set to the same value as the name of the layer (line eight). This transformation is executed for every layer found in the particular source model.

Of course, this transformation rule is not satisfying because it does not consider the architectural rule that applies to the layer and, hence, to the package. One possibility would be to create a corresponding constraint in the target model. It is possible in principal to create complex model structures in the target model with ATL, and it is also possible to attach constraints to model elements in UML, for example written in the *Object Constraint Language (OCL)* [Obj10a]. This is added informally to the ATL rule in Listing 2.2. Such as name, also ownedRule is an attribute UML packages possess. It contains possible constraints attached to a model element. The transformation definition depicted in List. 2.2 creates the informally denoted constraint for each package created for a layer (see lines 9–12). However, there are several problems with this solution.

The first problem is a technical problem with OCL, which is the de-facto standard constraint language for UML. Structural invariants like the constraint above are always formulated in the context of an instantiable model element but are evaluated for the instances. If a model, for example, specifies a class containing an integer attribute and an invariant that the attribute must always contain a value greater zero, the constraint will be evaluated in the modelled system for the single instances. It is not possible with OCL to define invariants in a model that restrict the

Listing 2.2: ATL rule to convert layers into UML packages with dependency constraint.

```
module Arch2Design
create OUT : UML from IN : Arch
rule Layer2Package {
  from
    l : Arch!Layer
  to
    p : UML!Package (
      name <- l.name
      ownedRule = 'Nothing in this package depends on
                   something contained in a package
                   created as result of the transformation
                   of a layer above l'
    )
}
```

model structure itself. But this is exactly, what architectural rules define. This missing feature of OCL has been considered an important drawback in recent research [Mat10].

The second problem is a conceptual one. Let us assume, that there is a constraint language, which we could use to define a constraint that is generated in the target model as in Listing 2.2. For every possible kind of artefact, defined by a meta model, a corresponding constraint had to be integrated in the set of transformation definitions containing a rule for transforming layers into whatever would represent layer in the kind of design artefacts; the constraint would have to be defined repeatedly for different meta models, causing overhead in maintaining the set of rules. Instead of having defined the set of architectural rules once, as Fig. 2.2 suggests, the same architectural rule is replicated several times in different sets of transformation rules and different target models.

It can be stated in conclusion that architectural compliance checking mechanisms are also required in MDSD due to the described deficiencies. Model transformations alone cannot address the task to check constraints, defined in one model, on a set of different models as instances of different meta models sufficiently.

2.3. Component-Based Software

This section gives a short introduction to *component-based software* also known as *component software*. Due to its wide spread of usage, this class of software systems will be subject to the overall theme of this work, the architectural compliance of software systems.

2.3.1. What is Component-Based Software?

The basic idea of component-based systems is that systems are composed of custom-made or third-party off-the-shelf components. The motivation of this approach is to combine the advantages of custom-made and standard software in order to minimize the disadvantages that both approaches bring if applied purely and alone. Custom-made software fulfils the customer specific requirements often very well because it can be precisely tailored to the customer's needs. On the other hand, it can be quite expensive and may underlie a longer time-to-market compared to standard software. Standard software products, in contrast, are only slightly tailored to customer specific needs and may be cheaper compared to customer specific solutions. On the other hand, it is in general not tailored to single customers as effectively as a custom-made solution; moreover, a standard product might not lead to a competitive advantage as easily as an extraordinarily well adapted custom made solution because business rivals might use the same standard product [Szy02].

Thus, component-based software development proposes to conduct a software system out of single components that can be either custom-made or from third parties. This allows combining the advantages of both worlds. Consider, for example, a system made of a number of standard components having the advantage that they do not have to be implemented by the system provider himself. On the other hand, the configuration and "wiring" of the components leave enough freedom to allow significant customer-specific adaptations.

The concept of components has been known for a long time in many engineering disciplines — an oft-cited example are integrated circuits from the electric engineering field. The idea to compose complex electronic circuits out of simple integrated circuits allows higher reuse of components and an increased productivity. Former approaches to apply the same concepts to the development of software are discussed in the introductive parts of the book on component software by Clemens Szyperski [Szy02].

The same book gives an informal definition of the term of a component, which is naturally the central construct in component-based software and always understood as *software* component in the following. The definition states important properties of components:

- Components are executable units.
- Components are units of independent production and deployment. They define by interfaces which functionality and properties they provide to and require from the environment and/or the surrounding system.
- Components can be composed to form systems.
- To enable composition, components conform to a certain component model and target a particular component platform. A component platform is a technical environment, in which a component can be deployed and executed (e.g. Enterprise Java Beans (EJB) [BM06]). Component models will be discussed in the following section.

In this work, we will focus on the component-based systems for two reasons. First, component-based system are today relatively wide-spread and can be found in many domains. One reason is the increasing distribution of software systems which rather consist of communicating components than of a single monolithic block. Examples of general component technologies accepted in industrial practice are (D)COM, .NET, the component model of CORBA, EJB and J2EE, Spring [Szy02], to name a few. EJB, for example, provides a component model for distributed enterprise applications. Application-specific solutions, like the Eclipse Platform [CR08] with its plugin principle, are basically component-based systems, too. Hence, focusing on this class of systems for the task of investigating architectural compliance checking should reduce some complexity of the overall problem without limiting a solution to a small and irrelevant set of software systems.

Second, the first initial thoughts for this work on architectural compliance have been caused by observations regarding an example of a component-based system and different approaches to describe it architecturally in an adequate way. Those observations resulted in the conclusion that component-based design massively suffers from the general problems regarding architectural compliance.

2.3.2. Component Models

To precisely define, how component can be executed, how they can be composed, and how they can communicate and interact, components for a certain platform have to conform to a certain component model. A component model consists on the one hand of a system model, which describes runtime properties that components have; on the other hand, it often provides a certain description technique to specify components and other relevant constructs.

System Models and Description Techniques

The system model of a component model describes runtime properties of components. In popular component technologies, the system model describes the technical framework in which components run. For example, the system model of EJB describes how the different kind of beans (how components are named in EJB) are instantiated, that they run in a container, and which and how the container manages accesses, the lifecycle of components, and so on. There are also formal component models, like DisCComp [Rau04] describing such runtime properties mathematically. In general, a system model describes properties like:

- How can components be composed? Are they typed and how can they be instantiated?
- Lifecycle of components, how are components created and destroyed, which states can they run through controlled by which circumstances? How are components deployed?
- Communication aspects. How can components interact? Which kind of communication is allowed, for example, asynchronous or synchronous communication, or both?
- How is concurrency addressed?

Systems to be build have to be specified. In different component-based development approaches, different description techniques are supported. EJB, for example, is a Java technology, hence components are specified by Java classes annotated with EJB-specific annotations indicating special kinds of components (entities, session beans, message-driven beans). For the graphical design of EJB applications, a special UML profile exists [Obj04]. The mentioned DisCComp component model provides a custom-made textual specification language [Rau04, AHKR08] as well as a graphical notation based on UML [AHKR08].

While the system model describes important runtime characteristics of the systems that can be build, the description technique massively influences how well the systems can be described, at which level of abstraction they can be described, and which properties can be influenced by the designer by specification.

Component Models by Comparison

In 2007, members of the component-based software development community initiated the CoCoME project[5]; the name stands for *Common Component Modelling Example*. More like a competition, the purpose of the project was to compare different component models from academia and to evaluate the state of the art in the field.

The competition was motivated by the fact that there is a large variety of component models differing in many aspects. Different approaches have different system models realizing concepts like components, interfaces, ports, etc. in different ways, and treat hence composition, communication, concurrency, and deployment differently. They use different description techniques, graphical as well as textual ones, with features for difference focuses, e.g. modelling of non-functional properties, and are based upon different system or run-time models.

The goal of CoCoME was hence to get an overview of the existing, confusingly multifaceted set of approaches to component-based software development and to compare their features, strengths and weaknesses. The idea was realized to define a common exemplary system which could be modelled by different component-based approaches. A common example allows the research community to compare and to validate new or existing approaches, and to focus research efforts to less intensively investigated aspects of component-based systems. At the end, thirteen teams from academia faced the task of the competition — the modelling of a common system with their own approach to component-based software development.

All of them were able to describe the system completely or in larger parts. Of course, the single teams focused on the main purposes and strengths of their approaches, like description of functional or non-functional properties, verification issues, seamless model-driven approach, and so on. However, as one of the results, one has to observe that none of the system models or the applied description techniques could be considered as being architectural. As we will see in Chapt. 4, which will present the system modelled in CoCoME, there were three architectural principles that the system followed. It seems that there has been a more or less common, informal understanding of these principles that has been intuitively followed in design and implementation of the system in the single teams. However, no description technique made those principles explicit to enable us to check architectural compliance.

This result emphasizes the need for a better integration of architectural design and detailed design for component-based systems.

[5]See also www.cocome.org

2.4. Research Questions of this Work

This work deals with architectural compliance checking under certain general constraints. First, it is assumed that a model-driven software development approach is followed in which all relevant artefacts are instances of meta models. Second, it is assumed that the systems under considerations are component-based software system.

Concluding from the problem analysis presented in this chapter, the main research question is formulated as follows:

> **Research Question 1 (RQ 1):** How can architectural compliance checking tool support be realized that is flexible with regard to
>
> - the large number of different meta models that have to be considered in checking architectural compliance (RQ 1.1), and
> - the variability and adaptability of different architectural rules that need to be checked (RQ 1.2)?

As we have seen in Sec. 2.2, a solution to RQ 1.2 has to answer the question which rules actually constitute architectural rules and how they can be described. Furthermore, we have seen that this question is closely connected to the question how architectural design and detailed design are distinguished. Hence, a second research question is formulated:

> **Research Question 2 (RQ 2):** How can models of architectural design, detailed design and implementation be distinguished from each other to clearly define a term of architectural compliance, which, in order to provide checking tools, can be checked algorithmically?

An approach that answers RQ 1 — which is more likely if RQ 2 is answered adequately, too — will significantly improve tool support for architectural compliance checking, which, at the moment, lacks flexibility in at least one of the two dimensions named above. The state of the art of architecture compliance checking will be discussed in Chapt. 3.

2.5. Summary

This chapter has investigated the open issues of architectural compliance checking. First of all, it has introduced an informal definition of the term software architecture and presented some of the most important tasks software architects have to perform. Among them, there is the task of controlling the detailed design and the implementation of the software architecture of a system and ensuring that the intended software architecture is correctly realized, or, in other terms, that design and implementation are architecturally compliant.

After that, the difficulty of ensuring architectural compliance has been illustrated, which is an effect of the inherent complexity. Due to the size and complexity of software systems, the task of compliance checking cannot be performed manually. Hence, an important issue is tool supported compliance checking. Flexible tool support has to deal with the complexity caused by the variety of architectural rules that exists and the multitude of different artefact types. We have also seen that this complexity and the resulting problems are only partially solved in MDSD approaches which are already capable of addressing many of consistency, conformance, or compliance issues in software development in general. Moreover, principles of component-based software systems have been introduced. It has been argued that most of the approaches to component-based design also suffer from the inability to express architectural principles and constraints. Finally, the two main research questions guiding this research work have been formulated.

In the following, we will see how available approaches to architectural compliance checking try to address the problems identified in this chapter.

Chapter 3.

State of the Art

Contents

The man who does not read good books has no advantage over the man who can't read them.

Mark Twain

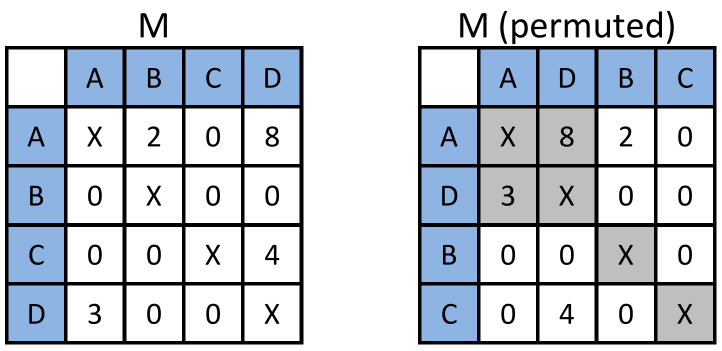
This chapter illustrates the state-of-the-art in the field of architectural compliance checking. It is structured according to a classification provided by [PTV+10], which distinguishes *Dependency Structure Matrices*, *Code Query Languages*, and *Reflexion Models* as different solution approaches; each of these classes will be presented in a single subsection providing short descriptions of representative implementations and tools. In addition, there will be a subsection about *constraint language-based approaches*.

After that, we will take a brief look on relevant neighbour research fields — architecture description languages and the formalization of patterns.

3.1. Dependency Structure Matrix-Based Approaches

Dependency Structure Matrices (DSM) are a technique to represent and analyse complex systems in general [Ste81]. Regarding architectural compliance checking, approaches basing on DSM represent the simplest form of checking approaches.

The main idea is to represent systems as a square matrix; each element that is considered as an element of the system, like components, modules, or subsystems, is represented as a


M

	A	B	C	D
A	X	2	0	8
B	0	X	0	0
C	0	0	X	4
D	3	0	0	X

M (permuted)

	A	D	B	C
A	X	8	2	0
D	3	X	0	0
B	0	0	X	0
C	0	4	0	X

Figure 3.1: DSM of a sample system consisting of four modules; the permuted matrix shows that *A* and *D* are cohesive and candidates for merging them into one single subsystem.

line and as a row of the matrix. The value e_{ij} denotes a dependency between the element i and the element j. Depending on the specific approach, entries are either binary (either 0 or 1), indicating whether there is a dependency or not, or any numerical value which in addition indicates the "strength" of a dependency; in this case, a value of 0 means that there is no dependency, a positive value indicates a dependency.

Figure 3.1 depicts this principle for the example of a software system consisting of four subsystems. The left-hand matrix shows that the subsystem *A* depends on subsystems *B* and *D*, *C* depends on *D*, and *D* depends on *A*. The values e_{ij} indicate the number of method calls from subsystem i to subsystem j. The software architect can easily see which dependencies exist.

Moreover, simple architecture analysis can be done manually or by tools based on analysis of matrices. The right hand matrix in Fig. 3.1 shows the result of applying a synchronous permutation of lines and rows to the left hand matrix, such that a filled square of cells is created symmetrically to the main diagonal[1]. This indicates a circular dependency between subsystems on the one hand, on the other hand a highly cohesive subset of subsystems which could be merged. As a further example, a matrix that has only entries in one half of the main diagonal indicates a system without circles. For more applications of DSM, refer for example to [Bro01].

Lattix LDM from Lattix [SJSJ05] is a tool to create, analyse, and control a software architecture in terms of allowed dependencies based on DSM. It uses a graphical high-level box-and-line notation to represent subsystems of a system, and is able to check violations of dependencies in Java bytecode, .NET code, Hibernate, Spring, and some more. Subsystems, as denoted in an abstract manner in Fig. 3.1 are basically Java packages, depend relations are usage relationships of any form existing in the artefacts mentioned above.

Basic architectural rules can be specified manually. There are basically two kind of rules: a "can-use" rule specifies a source and a target subsystem and expresses that the corresponding DSM entry is allowed to have a dependency value greater 0; a "cannot-use" rule specifies the opposite — the entry for a dependency between source and target must be 0.

Lattix LDM supports only a very limited set of architectural rules; basically, only rules regarding dependencies are possible. Furthermore, architectural rules are not captured in a

[1]The forming of such a matrix is also called *partitioning*, and the kind of matrix is also known as *block triangular matrix*.

software architecture model of the system but are only visible to the architect/designer/developer inside the tool. The rules have to be kept consistent to an existing architectural model manually causing significant maintenance efforts. The system's DSM is the main architecture documentation, not being able to illustrate the sources of rules explicitly, like for instance the layers that cause dependency constraints.

From the support for different architectural rules point of view, the flexibility of the tool is hence very limited. The support for different artefact types is appreciable, although the support for detailed design-specific artefacts is limited. Nevertheless the benefit should not be underestimated, since the understanding of existing dependencies is the first step to an understanding of the system and valuable, for example, for architecture recovery.

There are several architecture analysis tool that visualize actual software architectures by DSM but combine them in order to check intended architectures with query languages or reflexion models (see Sec. 3.2 – 3.4).

3.2. Query Language-Based Approaches

Query languages in general allow the specification of queries to retrieve information from a pool of data. SQL as the probably most popular query language retrieves data from relational databases (see, e.g. [EN10]). Source code query languages allow one to formulate queries upon a base of source code and retrieve single elements or more complex substructures from it. The purposes are diverse; queries can be used to check coding styles and conventions, searching for errors or possibilities to apply refactoring [KP07]. In model-driven development, such query languages are used to find the source patterns of transformation definitions or to check for consistency constraints. Naturally, they can also be used to define queries retrieving elements that adhere to or violate architectural rules.

The *Code Query Language (CDL)* [CQL] is an SQL-like language to query object-oriented source code. Many built-in keywords and instructions specific for object-oriented system allow easy-to-use and flexible specifications of queries and constraints. For example, a `SELECT METHODS` statement returns in combination with a `WHERE` clause defining a condition the set of methods in a system for which the condition holds. For example

```
SELECT METHODS WHERE isUsing A.doSomethingComplex()
```

retrieves all methods that call the `doSomethingComplex` method of class `A`. Constraints can be checked by prepending the clause `WARN IF <condition> IN <query>` which causes a warning if the condition is true for the result set retrieved by executing the query. Further clauses do not only allow references to structural dependency properties of source code, but also to metrics, test coverage, etc.

CQL is used in the tools *NDepend* [NDe] from SMACCHIA.COM and XDepend from OCTO Technology [XDe]. NDepend is a comprehensive source code analysis tool suite, including CQL for querying code and checking design rules. It supports all .NET programming languages and C++. XDepend is comparable with NDepend but covers the Java market.

Tools based upon CQL provide powerful support for architectural compliance checking. They are very flexible with regard to the different kind of rules. However, NDepend and

XDepend only support implementation languages but not modelling language for the detailed design. Moreover, there is no support to integrate the generation or definition of architectural rules directly into a software architecture description. Furthermore, component-based software and its descriptions are not considered by CQL.

A similar approach to CQL is *.QL* from Semmle [dMVH+07, QL]. .QL's syntax is also comparable to SQL and there are several built-in instructions to query object-oriented source code. .QL's support for custom-made queries is even more sophisticated that CQL's, since it is possible for users to specify their own predicates to be used in queries. This is done in a Java-like object-oriented language which should be familiar to the targeted group of users, namely software designers and developers.

The main critics of CQL apply also to .QL, i.e. the missing integration with software architecture models. .QL is integrated into Semmle's source code analysis tool suite *semmle/code* for Java.

JQuery [JV03, Vol06] is an approach based upon the logic programming system TyRuBA [Vol98]; TyRuBA provides a Prolog-like logic programming language which is typed, in contrast to Prolog [SS94]. JQuery uses TyRuBA to represent Java source code as logical knowledge base, i.e. as logical facts and derivation rules how to derive new knowledge from existing facts. As known from Prolog, such a knowledge base can be queried to retrieve data, in this specific case about the represented source code.

The prototype described in [DH09] applies JQuery to enable architectural compliance checking whereas architectural rules are embodied as logical queries that return violations of the corresponding rules. It was applied to check architectural rules in source code; a major drawback is that the intended architecture has to be entered manually as facts, because there is no integration with an appropriate ADL. Moreover, the selection of JQuery restricted checkable artefacts to Java bytecode and source code.

The research closest to the proposed approach is the logic meta programming approach proposed by Kim Mens [Men00]. In his doctoral thesis, he describes a framework to automating architectural conformance checking by representing architecture and implementation of a system by a logical knowledge base. A simple architecture description language allows describing a system as a structure of *concepts* and *relations* between concepts. *Architectural instantiation* describes how a model written in this language is transformed into *architectural abstractions*, which describe how concepts and relations are represented as statements of a logic programming language. The available predicates to formulate those statements are driven by a fixed implementation language (Smalltalk [Gol89]) and the *Smalltalk Open Unification Language (SOUL)* [Wuy01], a logic programming system representing Smalltalk source code by logical knowledge bases.

As later chapter will show, the presented work can also be understood as approach applying logic meta programming to conformance checking, although the problem in general is investigated on a more conceptual level; however, the implementation presented in Chapter 8 is based upon logic programming. On the conceptual level, the main difference is that in [Men00] only architectural descriptions are understood as logical statements *about implementation artefacts*, causing the mapping to depend on implementation artefacts, namely the structure of object-oriented system as understood in Smalltalk or SOUL, respectively. The proposed approach understands *every* description of a system, including the implementation, as statement about *a class of systems*, namely component-based systems, and maps descriptions onto a common ontology. Architectural rules, which are represented in [Men00] in the mapping, depend only on the structure of the ontology but not the implementation artefacts.

The implications are that the flexibility regarding the support of different artefacts to be checked is greater than in the approach proposed in [Men00].

Concluding, it can be said that query-based approaches are potentially very powerful. The expressiveness of all presented approaches allows us to cover a broad range of architectural rules. Some of the approaches are limited regarding the artefact types that can be checked — the flexibility ranges from single implementation languages (JQuery) to a whole set of implementation languages (CQL). Approaches for models in general are not yet used to realize conformance checks on detailed design model — it seems that architectural conformance is considered to be an issue that need to be checked on source code only.

The main drawback is that software architecture models are not integrated as source of architectural rules ([Men00] excluded to a certain degree); rules have to be specified separately and are not part of an architectural model of the system.

3.3. Constraint Language-Based Approaches

Similar to query languages, constraint languages allow the user to specify conditions that a pool of data must fulfil. The most prominent example is OCL which is widely used to specify such conditions in UML models, for example to specify invariants over class structures that cannot be expressed by class diagrams alone. Specified invariants have to be valid in all instances of the model, i.e. for all objects and the structures they form.

The disadvantages of OCL for architectural compliance checking have been already discussed in Sec. 2.2.2. Most approaches presented in the current section are languages to express and evaluate constraints over source code. Note that the principles of query languages and constraint languages are very similar in the sense that checking constraint can also be interpreted as querying elements that violate the constraint and checking if the retrieved set of elements is empty.

The Structural Constraint Language (SCL) is an approach to capture design intent as constraint over object-oriented program structures [HH06]. Its expressiveness compares to full first-order logic whereas the terms of formulae can use predicates and functions specific for object-oriented source code, like most of the query language-based approaches (see Sec. 3.2). Prototype implementations of the language are available for the Java and C++ programming languages.

A similar approach is *LogEn* [EKKM08] which has, compared to SCL, a less expressive language (no usage of quantifiers) which comes with the advantage of better performance in evaluating constraints, as the authors state. Moreover, it provides a simple graphical language to define constraints. An interesting feature of LogEn is the possibility to define *ensemblies*; these are units formed of sets of source code elements, abstracting from the single elements and providing a more coarse-grained view on the code. Dependencies are specified between ensemblies, allowing checking of rules, i.e. constraints, at different levels of abstraction. As the author state themselves, the main purpose is the checking of dependencies, hence other architectural rules are neglected.

The *Dependency Constraint language (DCL)* is another constraint language-based approach [TV09]. It defines a domain-specific language for the specification of constraints, significantly simplifying the language but restricting it expressiveness. The user of the language defines *modules*, which are sets of classes, and selects from predefined dependency constraints to express allowed or disallowed dependencies between modules. Different types of dependencies are allowed such as “A calls B” or “A creates B”, exemplary constraints are “can-create” to express that a create dependency is actually allowed between two modules. The approach is implemented by the *dclcheck* tool for Java.

The range of flexibility provided by query language-based approaches regarding the variety of architectural rules differs. While SCL is based on first-order logic with object-oriented predicates, LogEn reduces the expressiveness of the underlying logic, and DCL reduces possibilities for the composition of constraints. In all cases, the presented query language-based approaches seem to be more flexible. The support for different artefacts is limited to single programming languages which might be caused by the fact that most approaches are academic prototypes — industrial tool support is missing so far. The integration of an ADL and support for component-based systems is missing as well.

3.4. Reflexion Model-Based Approaches

Reflexion modelling was introduced in [MNS01] as technique supporting program and system comprehension. The assumed or intended software architecture of a system is modelled by a high-level model, containing elements and dependencies as they are expected to be. After that, a dependency graph is automatically extracted from the existing system artefacts, source code in most cases. The created graph is also called the source model. In the following step, a mapping between elements of the high-level model and the source model is created manually, capturing the “common” elements of the intended high-level architecture and the actual structure of the system. As a next step, the actual comparison of both is presented in a reflexion model, which depicts

1. dependencies present in both models as solid lines *(convergence)*,
2. dependencies present only in the source model as dashed lines *(divergence)*, and
3. dependencies present only in the high-level model as dotted lines *(absence)*.

The specific visualization differs from tool to tool but common is the separation of convergence, divergence, and absence. Hence, the reflexion model represents violations of architecturally intended dependency structures and possible mismatches. There are several approaches applying reflexion models for architecture analysis.

The Fraunhofer *Software Architecture Visualization and Evaluation (SAVE)* tool follows the reflexion modelling approach [LM08, DKL09]. The intended architecture of a system, the high-level model, is specified as UML model following the KoBRA approach to model components [MA02]. A source model can be generated from Java, C++, or Delphi source code. The tool has been evaluated in several case studies [KLMN06]. Current extensions of the approach deal with continuously checking of architectural violations instead of checking at actively initiated checks at dedicated points in time; architectural compliance is checked permanently in the background while software developers modify source code, giving them instant feedback and providing a constructive support to adhere to the architecture.

The tool *ConQAT* [DHHJ10] is very similar to SAVE, providing a high-level graphical modelling language and supporting a set of implementation languages. The authors emphasize the need and support of diverse artefact types which they try to provide designing their tool support as pipes-and-filter system which allows them to easily add filters for processing additional artefact types. This is noteworthy because it is not clear for most of the other approaches with limited support for different artefact types how they, or their implementations, can be extended.

The *Bauhaus* tool suite allows, amongst other analysis, reflexion modelling for Ada, C, C++, Java, and more programming languages [RVP06]. Representations of systems are either written in *Intermediate Language (IML)* for a detailed view or as *Resource Flow Graph (RFG)* for a coarse-grained architectural view. While IML is basically a implementation-language independent representation of source code, RFG are hierarchical graphs whereas nodes denote coarse-grained elements like types, components, files, etc., depending on the architectural view. Intended architecture and actual architecture are both represented as RFG for reflexion modelling whereas the RFG for the existing code is automatically generated.

The free tool *Dependometer* let the software designer describe an intended logical software architecture in terms of layers, slices (structure elements orthogonal to layers) and subsystems and map them to source code elements like packages or namespaces in Java, C++, or C# [Dep]. The architecture is defined as XML file. Checking generates a HTML report with navigation capabilities allowing software designers to browse between the intended architecture and the mapped structure in source code.

The flexibility of Dependometer is very limited in every dimension. The number of supported artefact types is low, the only architectural rules supported are dependencies defined by layers or slices. The definition of a software architecture in XML is very technical and not quite intuitive.

SonarJ [Son] and *Sotograph/Sotoarc* [BKL04] from hello2morrow are comparable to Dependometer. They structure systems architecturally also by layers and slices but provide a graphical modelling language for the purpose of specifying the architecture of the system. While SonarJ covers the Java market, Sotograph can also deal with C# and C++. Similar with regard to methodology and functional range is *Structure 101* from Headway Software [Str]. The criticism passed on Dependometer is also valid for the other tools of this group — neither the set of supported architectural rules is large, nor the supported number of development artefact types.

Concluding, there are interesting commonalities of the reflexion model-based approaches and important differences regarding their flexibility and applicability in the setting selected for this work. All have in common that a mapping between architecture and source code elements (in fact, they all check source code but not detailed design artefacts) is required to keep track how architectural elements are realized in code. In model-driven development, this information could be partially retrieved from persisted transformation information, such that manual creation of mappings could be easier or completely omitted. For example, consider a transformation definition can be specifying how to transform a component from an architecture description to a package of Java source code. If this transformation is actually executed, and tracing information is persisted, the same information can be used for reflexion modelling. Hence, this redundancy of information is a minor but not unsolvable drawback in the context of MDSD.

The single approaches illustrate that flexibility of compliance checking with regard to different artefact types is possible using reflexion modelling. For example, Bauhaus uses an abstraction of specific implementation languages. However, compared to query language-based approaches or constraint-based approaches, their expressiveness to architectural rules is limited because they do not allow the specification of arbitrary constraints about the structures they define. This is also due to the fact that the way architectural elements can be mapped to source code elements is limited by the expressiveness provided by the language defining the mapping, and the fact that not every kind of source code element can be referenced.

3.5. More Related Work

The presented related work so far is directly from the field of architecture conformance checking. Closely related are concepts of inconsistency management and model transformations in MDSD, as well as formalization of software architectures and architectural patterns.

3.5.1. Consistency in Model-Driven Software Development

As outlined in Sec. 2.2.2, among the main MDSD concepts are techniques to address architectural compliance checking rudimentarily. Two field are relevant in this context — (in)consistency management and model transformation.

Inconsistency management in MDSD deals with checking and ensuring consistency between different models or between views on the same model (when consistency is checked towards a common meta model). Moreover, inconsistency management is also about repairing inconsistencies. A survey of UML-specific approaches is given in [EB04, KHR+03, UNKC08]. In many cases, OCL is used to specify inconsistency rules, i. e. rules that describe when inconsistency constraints are violated, comparable to the negation of architectural rules describing when a detailed design model conforms to an architectural model. Architectural rules defined this way had to be defined for every combination of architectural meta model and development artefact meta model because the shape of the inconsistency constraints depend on the structures of the related meta models. The effort of maintaining architectural rules increases with the number of meta models whose instances could be affected by the architectural rules. The same problem also exists for approaches using other formalisms than OCL, like graph grammars and transformations, like [MSD06] or query-languages for semi-structured data like XPath in [NCEF02].

The same reason limit the applicability of model transformation approaches, as seen in the example of Listing 2.2, even if the disadvantage of OCL — constraints are evaluated on the instance level, relative to the modelling level a constraint is specified on — is omitted by choosing a different formalism like graph grammars (see, e.g. [BHLW07]).

From the theoretic point of view, the reason for the difficulties is the intensionality of architectural rules (see Sec. 2.2). The intension of an architectural element, i.e. its constraint restricting the further development, must be defined separately for each meta model.

3.5.2. Architecture Description and Formalization of Patterns

ADL are high-level description languages and have the potential ability to enforce architectural conformance by construction. However, they do not support this feature per se but depend on integrated development tools and code generators [MT00] or implementation frameworks [MMM05] — especially constraints for the refinement of architecture specifications are rarely supported. Hence, the same problems remain as in MDSD. An interesting approach in this context is ArchJava [ACN02] which extends the Java language with architectural language constructs. This simplifies the mapping of elements of the intended architecture to elements of source code. But beside compatibility issues to existing code [AAC07], and the fact that other artefacts cannot be accordingly annotated, the formulation of intensional architectural rules remains an open issue also in this approach.

Formal approaches to pattern formalization are also close to the field of architectural rules. Most approaches deal with design patterns [GHJV95]; a collection of approaches is contained in [Tai07]. The solution part of design patterns, describing in case of object-oriented design patterns a structure of classes, how they are interconnected and how they interact, also defines constraints for the participating elements. Hence, a precise formalization of design patterns has to express these issues. In fact, such constraints for architectural patterns [BMRS96] manifest architectural rules.

However, the main difference is that most approaches do not address the question how such a constraint defined in one model can be checked in another model, as need for the general checking of architectural rules. Most approaches, especially those that try to integrate their formalization into a MDSD framework like MDA, move pattern specifications and constraint definitions to the meta model level; constraint checks, e.g. written in OCL, are executed for the instances, i.e. models conforming to the meta model [BLE10, EBL06, FKGS04, KC09, MCL04, MHG02]. They are thus not applicable without modifications to the problem of inter-model constraints such as architectural rules.

3.6. Summary

It has to be stated that the state-of-the-art of architectural compliance checking in the context of MDSD does not provide powerful and flexible tool support. Architectural conformance checking cannot be realized with MDSD concepts alone in a satisfiable way. Since additional single tools do not constitute flexible solutions alone, whole tool chains are required. These are difficult to configure, to use and to maintain; potentially, they cause high acquisition costs.

The separation of software design into the three steps of architectural design, detailed design, and implementation is not considered in any of the approaches. Artefacts that are checked for conformance are source code of arbitrary programming languages, detailed design is ignored. The approaches do not distinguish between architectural design and detailed design. Another common property of most approaches is the focus on object-oriented systems, the modelling of component-based systems is only applied in SAVE on the architectural level.

The flexibility regarding the supported set of artefact types is difficult to judge. Just because a conformance checking tool supports only a single programming language to be checked, it does not mean that adopting it to other languages is impossible. To evaluate the flexibility, as long as it is not conceptually too strongly restricted, the adoption efforts had to be considered, which is difficult especially for commercial tools.

The most flexible approaches regarding the variety of architectural rules are query language-based and constraint-language based approaches. The expressiveness of those languages is only limited by the formalism they use, e.g. first-order logics, and the set of system elements and relationships that are represented in the formalism. Most approaches reflect object-oriented systems with elements like classes or methods and relationships like inheritance or call relations. In addition, query language-based and constraint-based approaches can be used very flexible to define user-specific and customized rules.

Reflexion modelling approaches have limited expressiveness and hence limited flexibility regarding architectural rules. They can only express constraints regarding dependencies, whereas most of the constraints follow a fixed and built-in scheme; the variation point that the user can influence is the specific mapping between architectural elements and source code elements. The same holds for dependency structure matrices.

However, reflexion model-based approaches at least potentially provide a high-level model of the system which can be understood as architectural model used to describe and communicate an architecture, e.g. SAVE. The presentation of an architecture as DSM is less adequate. Queries describing constraints on structures are often very detailed and lengthy; they are hence not adequate to communicate a high-level view on the system under consideration.

Concluding, it must be stated that no approach addresses all dimensions of flexibility sufficiently. In the following, the case study to which the proposed approach is applied will be introduced; readers interested in how the proposed solution addresses the flexibility issues might want to skip the chapter and to directly jump to Chapter 5, which describes the solution in detail.

Chapter 4.

Case Study

Contents

Few things are harder to put up with than the annoyance of a good example.

Mark Twain

This chapter describes the case study used to illustrate the need for a flexible architecture compliance checking approach. It contains a realistic component-based system and different application scenarios, in which compliance with the architecture of the system has to be checked. Section 4.1 introduces the *Trading System*, the subject of the *Common Component Modelling Example*, which will serve as case study. In Sec. 4.2, the rules and constraints will be examined that the software architecture of the Trading System imposes upon the design and implementation. Three modification scenarios, in which compliance checking is required to avoid architectural mismatches, will be developed in Sec. 4.3. These scenarios will be used later to validate the proposed approach (see Chapt. 7). The chapter at hand is summed up in Sec. 4.4.

4.1. The CoCoME - A Case Study

The application domain of the CoCoME (see also Sec. 2.3.2) is the retail industry. The functionality of the system covers the support of business processes common in retail stores like supermarkets. Specific use cases are, for example, the sale of goods and the management of the inventory.

At the start of the CoCoME project, the teams got certain input to start their development of the system. The use cases were handed out in textual form, supported by UML use case diagrams and refining UML sequence. Furthermore, a Java-based reference implementation of the system served as input. Class diagrams, component diagrams, and composite structure diagrams reflect the static logical structure of the system. However, to leave some freedom to special aspects of different approaches, they do not represent a full design specification of the system, but visually conclude the reference implementation.

The following subsections give a short summary of the CoCoME system, which is described in more detail in [RRPM08]. Section 4.1.1 describes the main functionality of the system and gives a top-level view onto the physical structure for a basic understanding of the system. In Sec. 4.1.2, the main architectural decisions will be explained. Section 4.1.3 will describe the UML design model of CoCoME. Finally, Sec. 4.1.4 concludes with some final remarks on the implementation.

More details about the motivation and goals of the project, the input for the teams, and further material can be found on the CoCoME website[1]. The approach-specific solutions and the evaluation of the competition results are described in detail in the CoCoME book [RRPM08].

4.1.1. Overview of the Trading System

The CoCoME system is inspired by the book of Craig Larman [Lar04], which introduces a fictitious "point-of-sale" system. The main use cases supported by the *Trading System*, this is how the CoCoME system is also called, are depicted in Fig. 4.1.

There are two functional areas. On the one hand, the Trading System supports the cashier at the cash desk of a store. The cashier scans product bar codes and handles the payment with the customer; on the other hand, the Trading System provides functionality for the back-office of a store. The overall system is hence decomposed into the *Cash Desk System* and the *Inventory System*.

The *Cash Desk System* supports the cashier in the process of selling goods and the payment. The cashier scans products, accepts cash or card payment, handles change, and so forth. There

[1]www.cocome.org

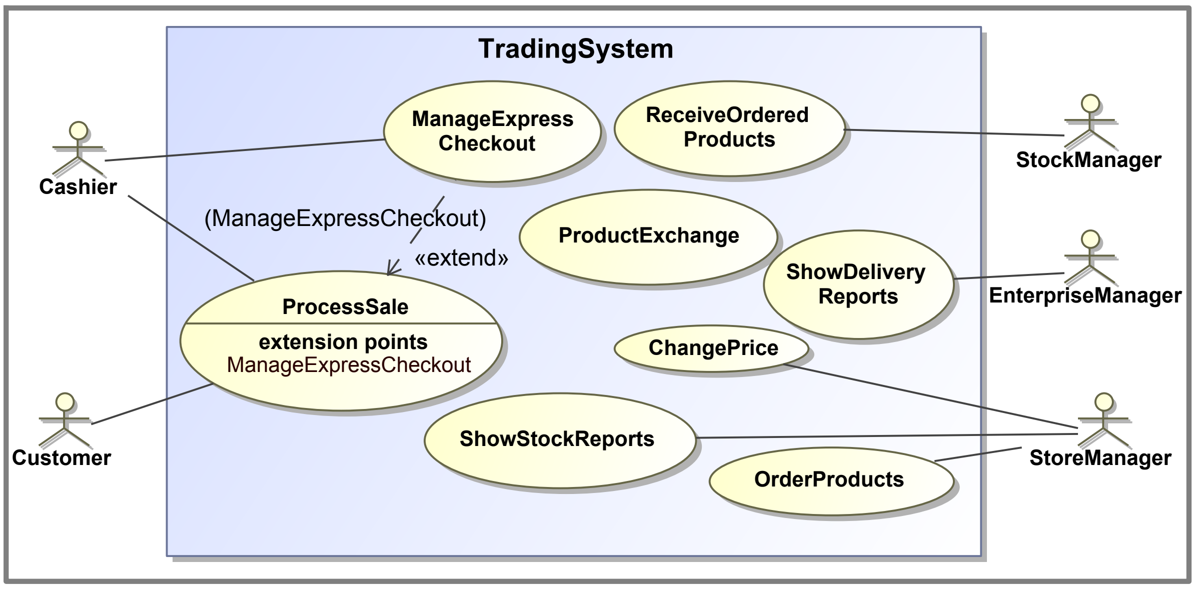


Figure 4.1: The use cases of CoCoME.

are many devices at a cash desk that interact during the process of selling and payment, for instance bar code scanners, cash boxes, and receipt printers. The interaction between these single parts is controlled by the Cash Desk Application software. The system allows the cashier furthermore to switch its cash desk to a kind of "fast lane mode" by analysing the latest payment processes and detecting a high percentage of small purchases. In this express mode, a LED sign is switched on indicating that customers with purchases up to a certain number of goods can be handled separately.

In addition, information about purchases is directly sent to the Inventory System. It updates the stock amounts of the products that are sold. This way, the stock is kept up-to-date permanently.

The Cash Desk System has a communication structure that is typical for distributed embedded systems [Mar05, BMRS96]. Independent, loosely coupled components interact by reacting to events that happen to some components in the system. For example, a cash box may only open if the cashier signals that he received cash, or a display showing the current running total has to be updated when another good is scanned.

The physical structure of a cash desk is depicted in Fig. 4.2. A central computer, which runs the Cash Desk System Software, is connected to the peripheral devices like cash box, bar code scanner, receipt printer, LED sign, and credit card reader.

The actors that interact with the *Inventory System* are different types of managers. The store manager is responsible for, among other tasks, setting prices and ordering new goods; the stock manager has to check in new goods.

The Inventory System supports the store manager by allowing him to change data of the product portfolio — the Change Price use case is one representative for similar tasks. The store manager can furthermore inspect stock reports. The system can report goods whose stock amount is below a defined, critical threshold indicating that new goods have to be ordered.

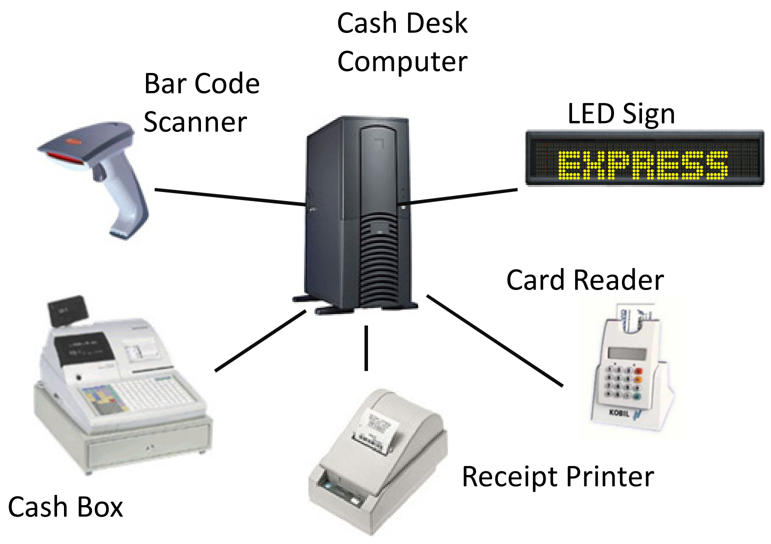


Figure 4.2: Cash Desk System overview (from [RRPM08]).

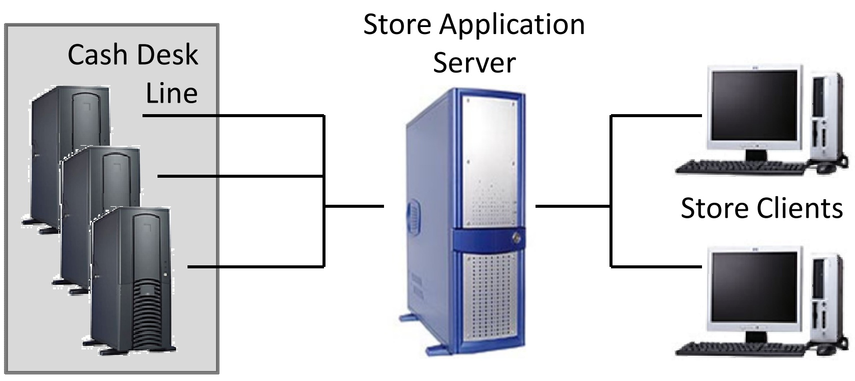


Figure 4.3: Top-level structure of the Inventory System (from [RRPM08]).

The stock manager can check incoming deliveries for consistency with the placed order by querying the Inventory System. Furthermore, he uses the system to check-in received goods electronically.

Figure 4.3 depicts the physical structure of the Inventory System. Central building block is a store application server, which has access to a database to read, update, and delete relevant data, like product data, order information, and so forth. Store and stock manager access that data via client computers connected with the server by a network. Furthermore, the server is also connected to the computers of the single cash desks in the cash desk line; remember, that data of the purchase processes is used to update the inventory.

Concluding, the inventory system is a typical distributed, client-server information system [TvS08]. Users read, update, and delete data persisted at a server, using client computers. In contrast to the Cash Desk System, the main communication mechanism is synchronous communication.

4.1.2. Architecture

The main architectural decisions and aspects of CoCoME were documented textually; an explicit architectural model was not given. Instead, detailed UML component diagrams were handed out as documentation of the system's logical code structure.

The main architectural aspects are the following:

Three Layer Information System. The Inventory System is a typical information system as explained above. One common reference architecture for systems like that is a Three-Layer-Architecture [Fow02]. It separates the components of a system into three distinct, hierarchical groups, the layers. Dependencies between layers are only permitted from "upper" layers to "lower" layers [BMRS96]. In general, layering decouples system parts of different levels of abstraction, or of fields of tasks. This leads to loosely coupled and easily exchangeable subsystems — for example, in a well layered system, changes in the upper layers will not affect the lower layers.

The lowest layer, often named "persistence layer" or "data layer", in this reference architecture has the tasks of persisting data and providing basic creation, access, and manipulation operations on the persisted data. For example, the persistence layer of the Inventory System contains a component Store, which defines entities for store and stock relevant data, for example StockItem or ProductOrder. Furthermore, the layer contains an interface StoreQueryIf defining methods to query the database for data; for instance, queryLowStockItems() retrieves all stock items that are running low (see also Sec. 4.1.1).

The layer above the persistence layer is the application layer and contains the application logic of the system. It provides complex functionality accessing the persistence layer to create, read, update, and delete data. For example, the application layer of the Inventory System contains a component, called StoreApp, which provides methods to order products and roll in ordered goods. These are complex operations consisting of several calls to data manipulation operations defined in the data layer. Only the application layer is allowed to access the persistence layer.

The layer at the top is the graphical user interface (GUI) layer. Its task is to interact with the users of the system and to present the output of the system to them. The components in this layer represent elements of the GUI. They interpret user input and call functionality of application layer components. In a three-layer information system, the business process realized by the system is embedded into the GUI layer, too. Simplified, each component, defines which functions are called to retrieve or manipulate data, and which other GUI components are invoked if its own control elements are used. This way, the control flow between components gradually reflects the business process. In contrast to four layer architectures with dedicated process layer and workflow engine [vdAvH04], there is no decoupling of representation and process control.

The GUI layer is not allowed to access the persistence layer directly. Concluding, the layer structure is *strict* — this means each component may only access components of the same layer or components of the layer *directly* below its own layer. The layering provides a separation between representation of data and control flow, application logic which works with the data and reflects single steps of the business process, and the data structures that contain persistent data.

Application Layer Interface with Copy Semantics. The layering of an information system structures the system logically. If the system is going to be implemented as distributed client-server or multi-tier system [TvS08], there are different possible variants to distribute logical layers onto physical nodes [TvS08]. In distributed systems, data as well as method calls between layers have to be transmitted between processes. If a system is realized in an object-oriented language, there are certain pitfalls regarding the transport of data, for example between the application layer and a GUI layer located at different. Let us assume, that a method call at the application layer returns remote references to objects to a caller in the GUI layer. Most probably, the caller will invoke methods at these references to retrieve the returned data. This kind of communication can be very expensive; each call of a remote object's method, like reading object attribute values or navigating references to associated objects, is translated to a remote method invocation via some middleware including marshalling and unmarshalling of calls, etc. This is very time-consuming and requires relatively large amounts of network resources compared to a simple transfer of data [Fow02].

A common solution to this problem is the use of *data transfer objects* [Fow02]. Instead of passing references to the network of persisted objects to the GUI layer, the required fragment of this network is copied into transfer objects. These copies are transferred to the GUI layer via the network; the GUI, running at a remote client computer, gets its own local and hence efficiently accessible copy of persisted data. Hence, we also say, that a component handling the passing of data the way described, provides an interface with *copy semantics* (in contrast to *reference semantics*).

Interfaces with copy semantics are an essential concept of service-oriented architectures (SOA) [KBS04]. The application layer of the Inventory Layer is designed as service-oriented system. This means that interfaces accessible from the outside of the layer, i.e. from the GUI layer, have to use transfer objects. A more detailed look at the implications of using data transfer objects will be given in Sec. 4.2.2.

Event-Driven Architecture. As mentioned above, the components of the Cash Desk System interact by causing and reacting to events. The event of scanning a product bar code (which is raised by the bar code scanner) causes reactions of other components — the running total is updated and visualized on the cash box display; the receipt printer prints another entry of the sale. Such a system is also said to have an *event-driven architecture* [YPMT09]. A common solution to realize an event-driven architecture is to use the *Event Channel* pattern [BMRS96], a special variant of the Publisher-Subscriber pattern [GHJV95].

Each component in the Event Channel Pattern can play the role of a publisher, a subscriber, or both. A publisher creates events and puts them to the channel, while a subscriber reacts to events that are put on the channel. When a publisher wants to "raise" an event, it calls a method at the channel which instruct the channel to publish the event. After the event is published, it is sent to all registered subscribers. They can react to the event as intended. The forwarding of the event does not necessarily happen at once after it has been raised by the publisher; raising and publishing are temporally decoupled.

An event-based architecture leads to very loosely coupled components and an easily extensible system. Since putting the event onto the channel and publishing the event are decoupled, the event channel pattern realizes asynchronous communication between a set of components that do not necessarily need to know each other.

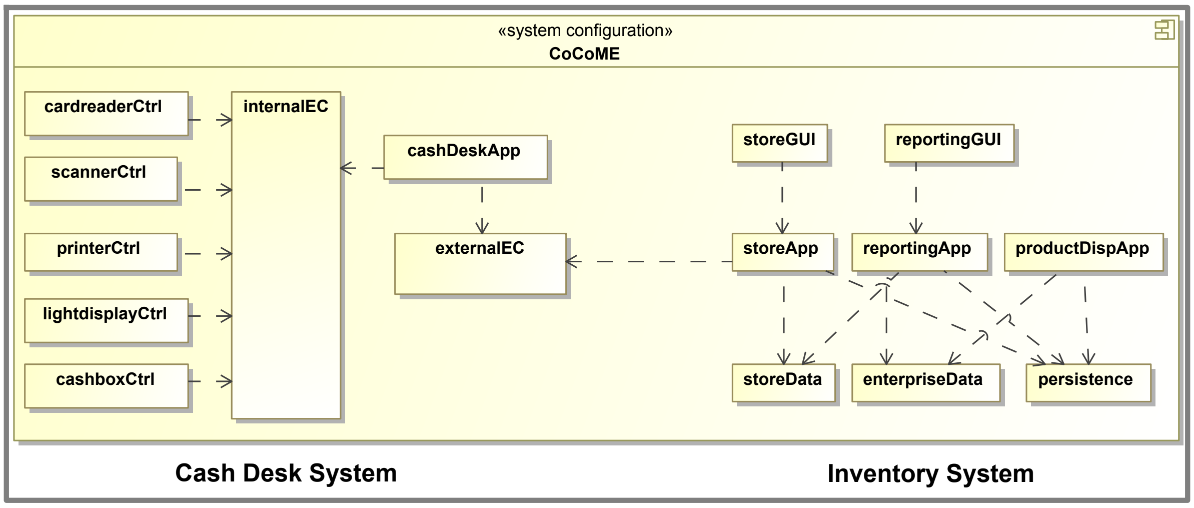


Figure 4.4: CoCoME design (top level).

In the Cash Desk System, there are actually two kinds of event channels. Each single cash desk has a separate channel internalEC connecting the software controllers for the single devices; the channel is for the communication among them and the communication with the cash desk application. For the whole cash desk line, there exists an event channel externalEC, to which the single cash desk applications register as publishers; the already mentioned application layer component StoreApp of the inventory system registers as subscriber. The cash desks publish “SaleRegisteredEvents” onto this channel, indicating that a sale process is finished. As a reaction to this event, the inventory system updates the amounts of goods in the inventory.

4.1.3. Design

The main diagram types of UML to describe the static logical architecture of a system are class diagrams, component diagrams, and composite structure diagrams. Class diagrams are used to model interfaces of components in detail, while component diagrams provide black-box views to components; composite structure diagrams show white-box views. Most of the UML language elements used in one of these diagrams types can also be used in the other two types, hence, the distinction between diagram types is not very precise (see [Obj10b]). The reference UML design model developed for the Trading System, however, uses these three diagram types.

Figure 4.4 shows a composite structure diagram of the overall Trading System, and all specified components. The single interfaces are omitted for clarity reasons.

The left-hand part of Fig. 4.4 depicts the Cash Desk System. There are components that represent the software controllers for the devices installed at a cash desk. Furthermore, there are components containing the cash desk application logic and a GUI component. They are connected via an internal event channel component.

The right-hand part depicts the Inventory System. The layered structure can easily be seen. In both, GUI and application layer, there are components for store-specific and reporting-specific functionality. As mentioned earlier, the store-specific components provide the functionality of

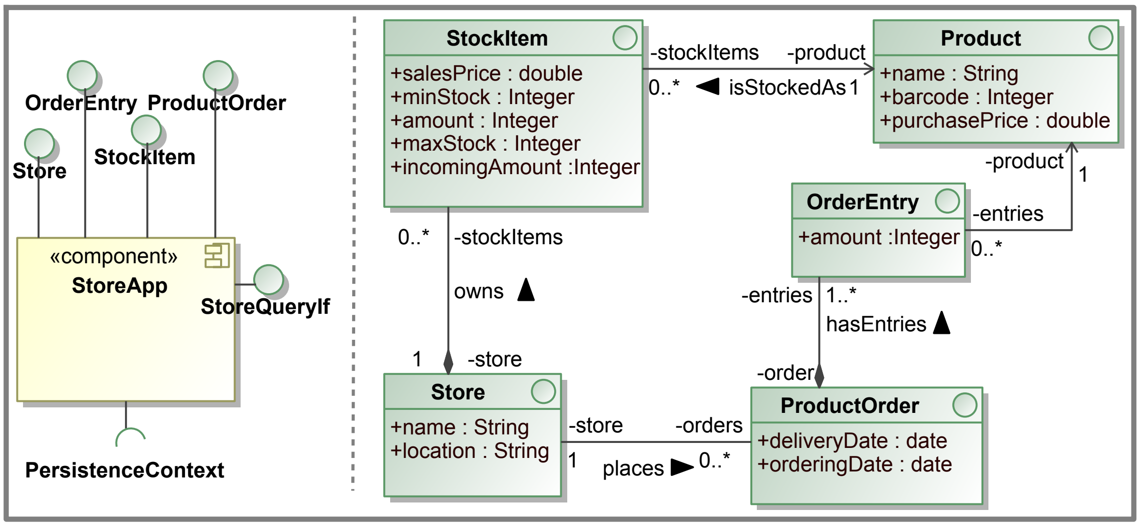


Figure 4.5: Component of the Trading System and details of provided interfaces; example from the data layer.

ordering products, changing product data, and so forth, at GUI level as well as pure application logic. Reporting functions are realized in a separate component. At persistence layer level, there is only one component Store, which contains the data model of the system and basic operations on the data model. The component Persistence encapsulates the technical realization of database access.

Figure 4.5 shows combined diagrams that model the Store component from the data layer of the Inventory System. In Fig. 4.4 was modelled, that there will be one instance of this type in the system. The left-hand part of Fig. 4.5 shows as a composite structure diagram depicting the inner structure of an instance of Store. The component provides several interfaces; StoreQueryIF defines methods to query the data persisted, e.g. all products available in the store; the other interfaces represent the entities of the data model, for instance Product and Order. The right-hand part of Fig. 4.5 depicts a class diagram of the data model showing the attributes of the entities and the associations between entities.

The component also defines a port with cardinality one, indicating that other components requiring an instance of type StoreQueryIf can connect to an instance of Store.

UML defines different types of behaviour diagrams like sequence diagrams, activity diagrams, interaction diagrams, and some more. They focus upon different behavioural aspects, for instance, sequence diagrams and interaction diagrams are well suited to model communication issues.

As we will see, most architectural rules that are considered here deal with communication aspects. Since a full design specification of the Trading System is out of the scope of this work, only communication between components will be modelled but not algorithmic details of methods. Hence, sequence diagrams will be used, that provide appropriate means to model the interaction between components.

Figure 4.6 shows the sequence diagram for the method getProductsWithLowStock. The corresponding operation is defined by the interface StoreIf, for which the component StoreApp,

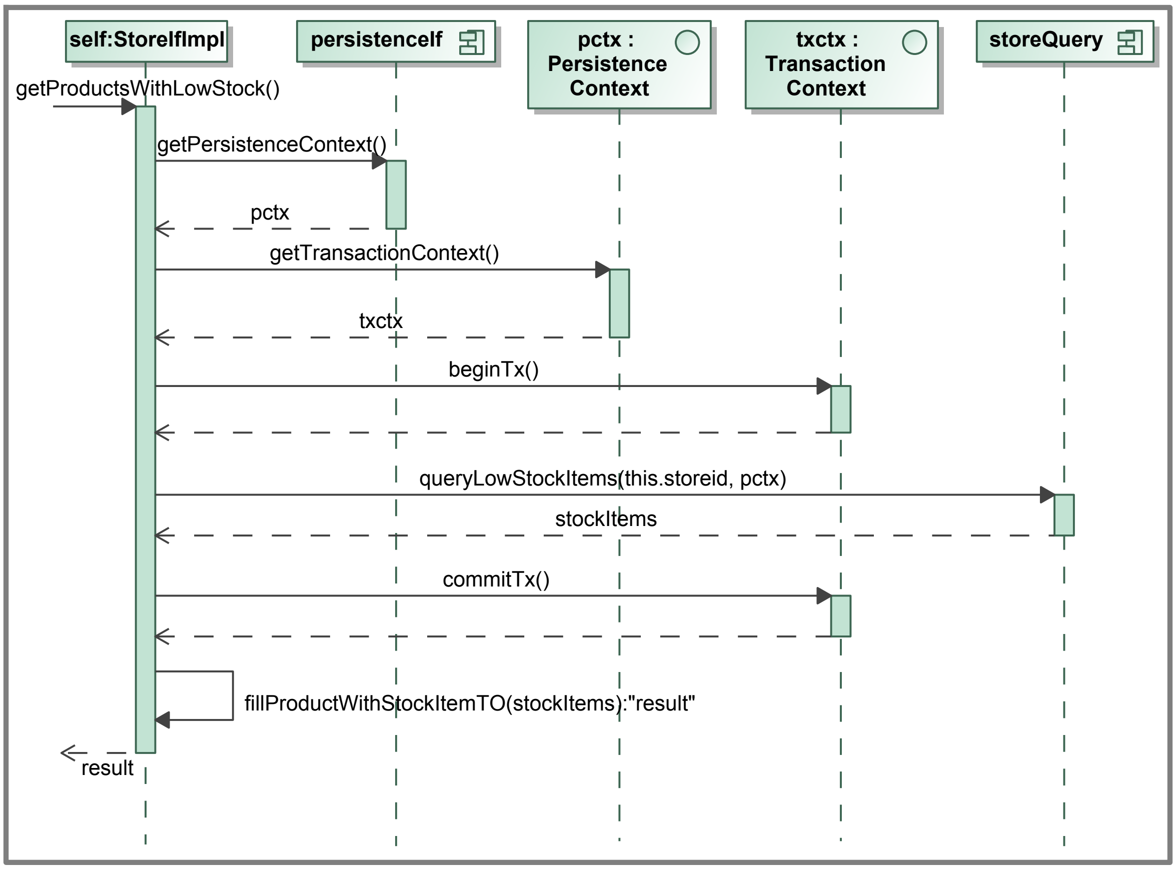

self:StoreIfImpl
persistenceIf
pctx : Persistence Context
txctx : Transaction Context
storeQuery
getProductsWithLowStock()
getPersistenceContext()
pctx
getTransactionContext()
txctx
beginTx()
queryLowStockItems(this.storeid, pctx)
stockItems
commitTx()
fillProductWithStockItemTO(stockItems):"result"
result

Figure 4.6: Behaviour specification for methods; exemplary sequence diagram for getProductsWithLowStock.

located in the application layer, provides an implementation. This implementation calls methods at the persistence interface first, which is modelled as a required port, called persistenceIf. The method calls are required to establish a connection to the database and to get a transaction context. After that, the query interface (also modelled as a required port of the application layer component Store) is used to execute a query retrieving the corresponding data. After that, a helper method fillProductWithStockItemTO is called to copy the data into transfer objects.

4.1.4. Implementation

The reference implementation, which was also given to the teams as input, had been realized in Java. The source code consists of 120 classes and interfaces, or about 6,000 lines of code. It was mainly used by the teams as a detailed reference behaviour specification, since the method bodies naturally contain a complete behaviour description, and to get a better understanding of the system by a runnable implementation.

During implementation, the developers paid attention to conform to the documenting UML diagrams, especially to map components systematically to source code structures in Java, which does not come with component-based concepts. Details regarding the implementation can be found in the documentation of CoCoME.

Technologies that were used for the implementation included Hibernate as object-relational mapper and persistence mechanism [BK06] and the Java Messaging Service as a technical off-the-shelf implementation of event channels [MRC09].

Hence, the implementation included more than only Java source code. Technologies like Hibernate and JMS include declarative configurations, definitions, etc. Hibernate, for example, allows XML Mapping Files to define how classes are mapped to tables of a database. They also define how associations between classes are mapped to foreign key relationships. These features are noteworthy because these artefacts can also be affected by architectural rules. Consider, we define that the data model of the Trading System is separated into two subsystems, one containing classes for data that is store-specific, one for data which is global to the enterprise, whereas the first subsystems may not depend on the latter. In this case, this architectural rule defines constraints that must not be violated by the definition given in the mapping files.

4.2. Examination of the Architectural Rules in CoCoME

In Sec. 4.1.2, the main architectural aspects of the Trading System have been described. They imply rules that guide the further refinement of the system during the design and the implementation. This section discusses those rules and describes them informally.

4.2.1. Layered Information Systems

Layers partition the set of components of a system, or a subsystem, into groups. Each of these groups contains components that logically belong together, because they are on the same functional abstraction level, abstract from the same more low-level details, or similar reasons. As [CBB$^+$10] states, each layer constitutes a *virtual machine providing a cohesive set of services* which may only use layers below itself.

But as described for the layered architecture of the Inventory System (see Sec. 4.1.2), this constraint is not enough to define every kind of layered architecture because it is not always desirable to allow a layer to use each of the lower layers. In addition to the concept of layers, a *allowed-to-use* relationship is required, modelling which layers in a system may actually use which other layers. Fig. 4.7 depicts a box-and-lines diagram style commonly used to illustrate layered architectures.

So far, we can conclude the following constraints for layers in general. A layer L may only use a layer M if

- L equals M *or*
- L is at an higher level than M *and* (L,M) is in the allowed-to-use relation

Hence, a layer is correctly designed and implemented if all of its usage relations are compliant with these constraints. However, it still has to be clarified what it means for a layer to "use" another layer. As a first refinement, a layer L uses another layer M, if one of the entities grouped in L uses an entity in M — these entities are components and interfaces, and everything that

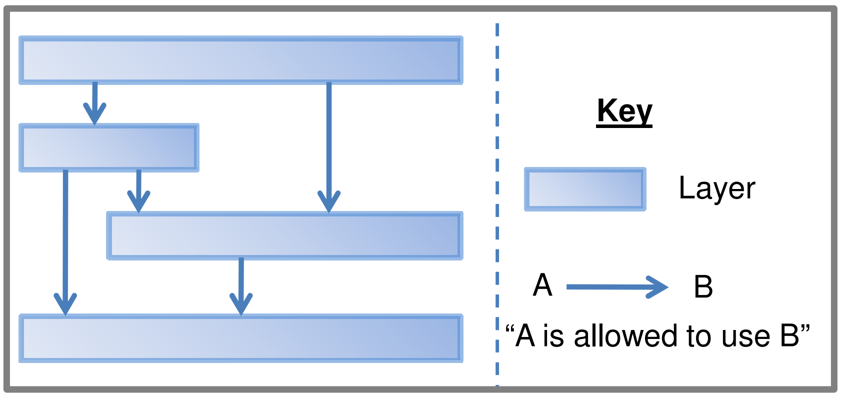


Figure 4.7: Informal layers documentation.

refines them in more detail, like classes, methods, parts, ports, connectors, and so forth. According to [CBB+10], a system element A uses an system element B, if A's correct operation depends on the correct implementation of B. Thus, if A is contained in layer L, and B in layer M, L uses M.

To make the definition of when the design or implementation of a layered architecture is compliant, complete, it hence has to be defined which cases of usage exist:

- *Usage between components and interfaces.* If a component provides or requires an interface, it assumes that the signature of the interface is what it is going to implement, or what it requires from its environment. In other words: changes of the signature imply changes of the providing or requiring components.

- For the same reasons, *specialization between types* leads to a usage relation between the specializing and the generalized type.

- *Usage of types.* An interface using a type, e.g. another interface, as method argument type or return type, or as type of a reference, depends on that type.

- *Method calls.* If the implementation of a component calls a method at another component (e.g., connected via a connector to a required port), it depends on the invoked behaviour.

The last case above needs some closer observation. In the case, a component invokes a method asynchronously, the invocation does not manifest a usage relation in the sense mentioned here. It's own implementation, which specifies the asynchronous invocation, does not depend on the correct implementation of the receiver of the invocation. Contrarily the synchronous call: the called method, for example, could incorrectly fail to terminate, causing the caller to wait forever.

Notice, that this definition allows layered systems to communicate in the opposite direction of the allowed-to-use relationships, at least to a limited degree, by asynchronous messages. This is often required for callback mechanisms, used for updates in patterns like MVC [BMRS96] or Observer [GHJV95], or to "bubble" error messages up to higher layers [CBB+10].

These rules have implications for the design and the implementation of the "Trading System" in CoCoME. Consider, for example, the components of the GUI layer. They are designed and implemented in compliance with the architecture (at least regarding the layered structure), if all they depend on is located in the GUI layer or the application layer, but *not* in the persistence layer. This means that GUI components or interfaces may not rely on an interface of the persistence layer nor be connected directly to a component of that layer nor call methods defined there. The same applies analogously to application layer components that may not be designed to use GUI layer components, and persistence layer components which may only use components inside the same layer.

Figure 4.8 shows this by example. It depicts on the left hand side a simplified architectural model describing the layers of the CoCoME Inventory System and the allowed usage relationships. The arrows stereotyped with «isAllowedToUse » model these relationships. Furthermore, the model describes how packages are assigned to layers. The package GUI is assigned to the layer called GUI Layer. On the right hand side, there is a cut-out of the CoCoME design model which models, among other aspects, the actually existing dependencies. For example, the component StoreGUI requires the interface StoreIf, which provides application logic methods relevant to the management of a store inventory. This interface is contained in a package that is assigned to the application layer. This is modelled by a «uses» relationship between the component and the interface. According to the architecture, this dependency is allowed because StoreGUI is located in the GUI layer and accesses an interface in the application layer. In contrast, the dependency towards the interface StoreQueryIf is *not* architecturally compliant. The architectural model does not contain a relationship which allowing a dependency between the GUI layer and the data layer.

The first application scenario will deal with the violation of the Trading System layer structure (see Sec. 4.3.1). It describes the addition of a component to the Inventory System that must be accessible from all layers.

4.2.2. Service-Oriented Interfaces

As mentioned in Sec. 4.1.2, the application layer of the Inventory System is designed to use data transfer objects to pass data from the GUI layer to the application layer and vice versa. These objects contain copies of data that is persisted in the data layer below. On the one hand, copying the data avoids direct access to the data layer, on the other hand it reduces network traffic because the number of remote method calls is reduced — instead of initiating many method invocations to navigate between objects, the relevant part of the object network is copied and transferred by a single method call (see. Sec. 4.1.2 for details).

The details of the concepts can be informally described by the following rules which follow the terminology from [HRB+08]:

- The GUI layer may only connect to interfaces that are declared *service interfaces*. Connectors that point inwards the layer, i.e. connectors that can be used to call methods at application layer components, are only allowed if they point to ports providing service interfaces.

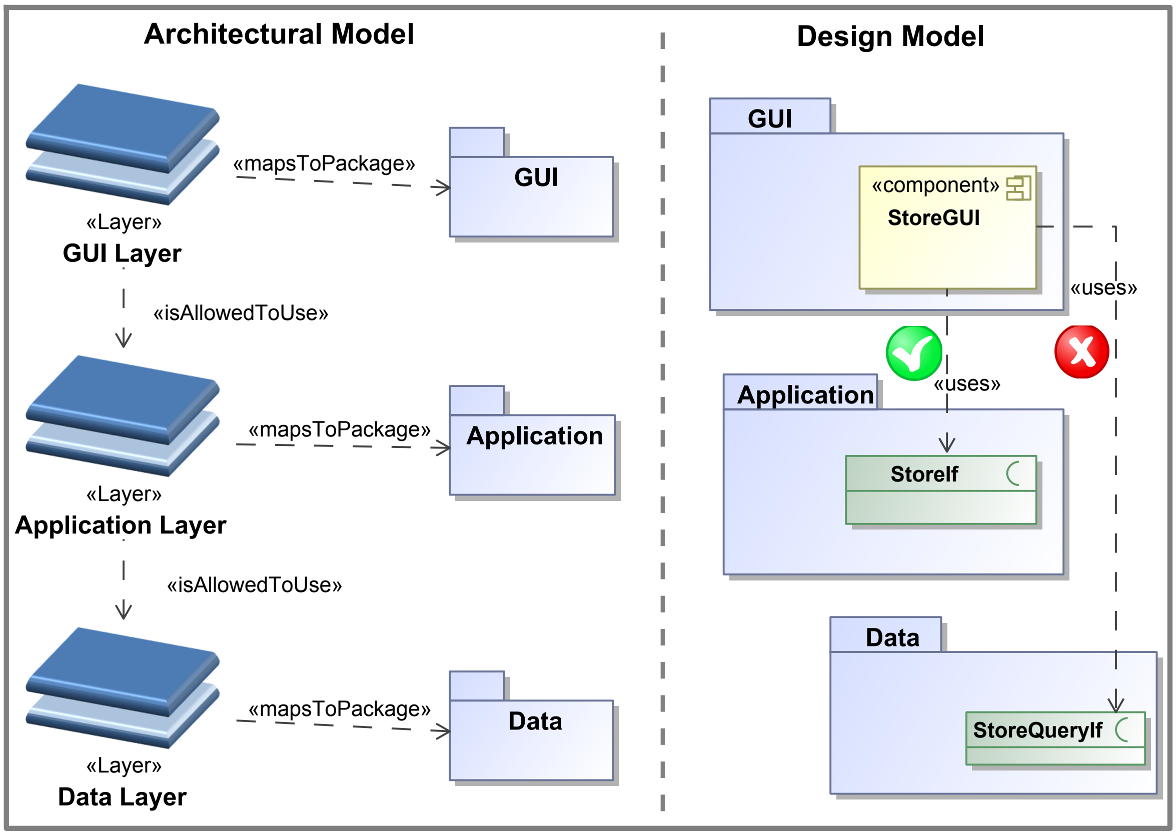


Figure 4.8: Modelling the layers structure of the Inventory System and compliant and violating dependencies in the design model.

- A service interface is an interface providing *service methods* only. Those methods use only *transfer object classes* or primitive types as parameter types or result types. Each call of a service method returns a new copy of data, i.e. a new instance of the transfer object class. This ensures that two different clients do not share a single instance as returned value.

- A transfer object contains only attributes of primitive types or refers to other transfer objects; it never refers to an object which is *not* a transfer object. Especially objects from the domain model are not allowed.

Different violations of these rules are possible. Figure 4.9 shows a GUI component that, on the one hand, correctly uses the services of the interface StoreIf. This interface, which is defined in the application layer, has only methods using only transfer object classes and primitive types. As a representative, the method getProductsWithLowStock is depicted, which returns an instance of the transfer object class ProductWithStockItemTO. On the other hand, the GUI component uses the application layer incorrectly. It accesses the interface OptimizationSolverIf, which is defined in the application layer and implemented by the application layer component ProductDispatcher (see Fig. C.12). The depicted method of this interface is not a service

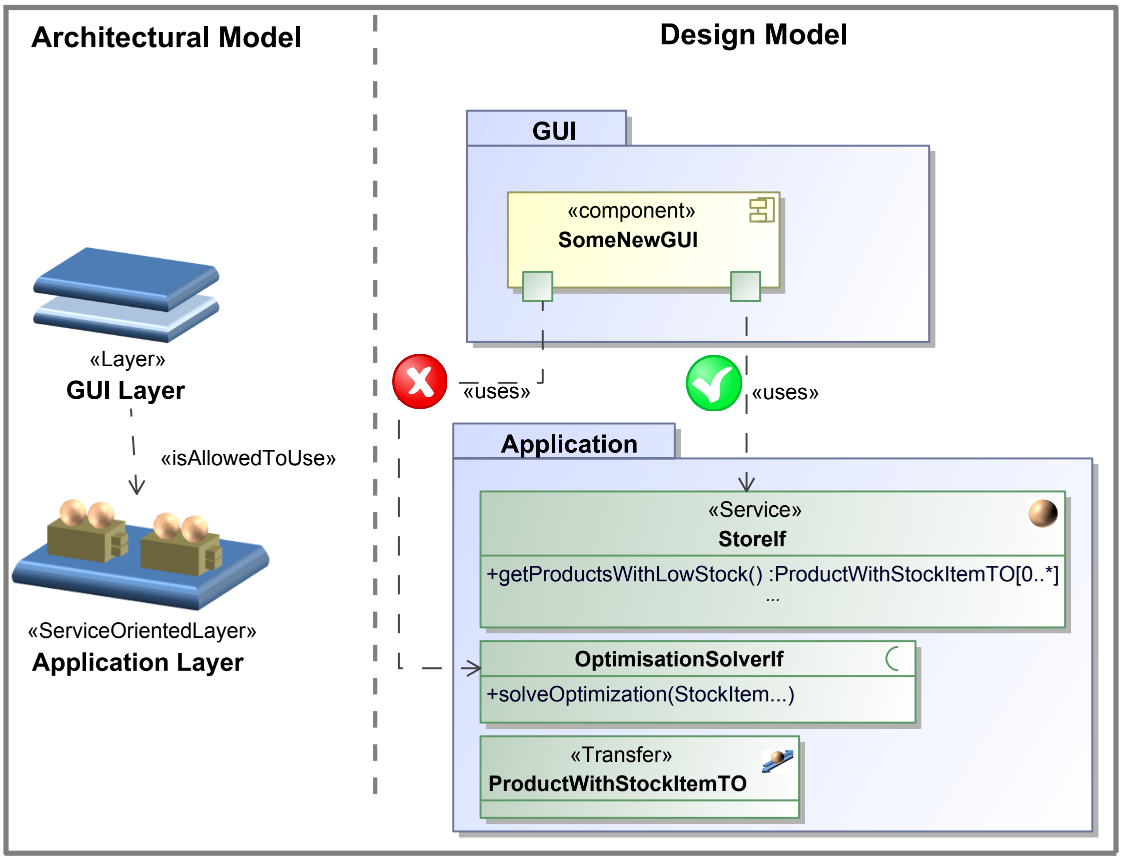


Figure 4.9: Examples of interfaces conforming to, or violating the service-oriented architecture of the Trading System.

method because it uses other interfaces than transfer object classes as parameter types; the parameter type StockItem is an entity from the persistent data model.

Notice, that the goal of the rules for transfer objects and interfaces with copy semantics is not to specify transfer objects completely. It is not constrained, for example, that a transfer object must contain the same data values of the relevant domain model extract. Instead, we describe a minimal set of rules that must be followed.

4.2.3. Event-Based Architecture

The event-driven architecture of the Cash Desk subsystem describes that the components in this subsystem communicate asynchronously and are further decoupled by event channels. The event channel has the task to forward events. Events are types whose instances can contain any relevant data about a specific event. They are modelled as interfaces and can also be specialized. Events are passed to the event channel and forwarded to all registered subscribers in an asynchronous manner. The components determine themselves whether to react to an event or not.

This means that the following rules apply:

- There are two kinds of components in the considered subsystem: event channels and participants; the latter are publishers or subscribers (or both).
- Components interact by raising events, modelled as a specific kind of type, and reacting to them.
- Participants connected to the same event channel are not allowed to communicate directly with each other using the events that should be published at the channel.
- Participants commit events to the event channel which ensures that these event are forwarded to the other participants at some point in time. This happens decoupled from the original commit call to provide a asynchronous communication between participants.

These rules try to avoid that the weak coupling between the components of this subsystems is undermined by directly "wiring" two or more components, as depicted in Fig. 4.10. In this example, all three components, cashboxCtrl, lightDisplayCtrl, and cashDeskApp, are plugged into the same event channel according to the architecture model. This is refined correctly in the design model specifying that those components are connected via ports to the event channel. But there is also a direct connection between cashDeskApp and lightDisplayController. Such a structure allows two components to communicate directly; they could exchange notifications of events without sending them to the event channel. Other components, that have subscribed for the same kind of events, would not be able to react to those events. This has to be avoided. In the example, the cash desk application could notify the controller of the LED directly about the activation of the fast lane mode; the controller can react and activate or change the text displayed on the LED. If the event is not published on the channel, the card reader component is not notified, and cannot deactivate itself as required in fast lane mode (see [RRPM08]).

Notice, for later sections, that communication infrastructure components like event channels are often taken "off-the-shelf". This also means that their behaviour and inner structure is often not modelled explicitly. This is also the case for CoCoME using JMS (see also Sec. 4.1.4). The correct behaviour is often taken for granted. To illustrate the approach to architectural compliance checking, a very simple event bus will be modelled to the degree that is required to show that the rules modelled above are followed or violated.

4.3. Detailed Modification Scenarios

Based on the reference implementation, a reference design model has been developed. In Sec. 7.1.1, an architectural model reflecting the three described architectural aspects including the architectural rules they defined, will be developed. The design model will be checked for compliance with the architecture model.

Since the reference implementation, and hence the design model, can be expected to be largely compliant, the approach will also be evaluated by scenarios introducing architectural violations into the design model.

In the following, three modification scenarios are introduced, each leading to a system design model that is not compliant with one of the three architectural aspects described in Sec. 4.2.

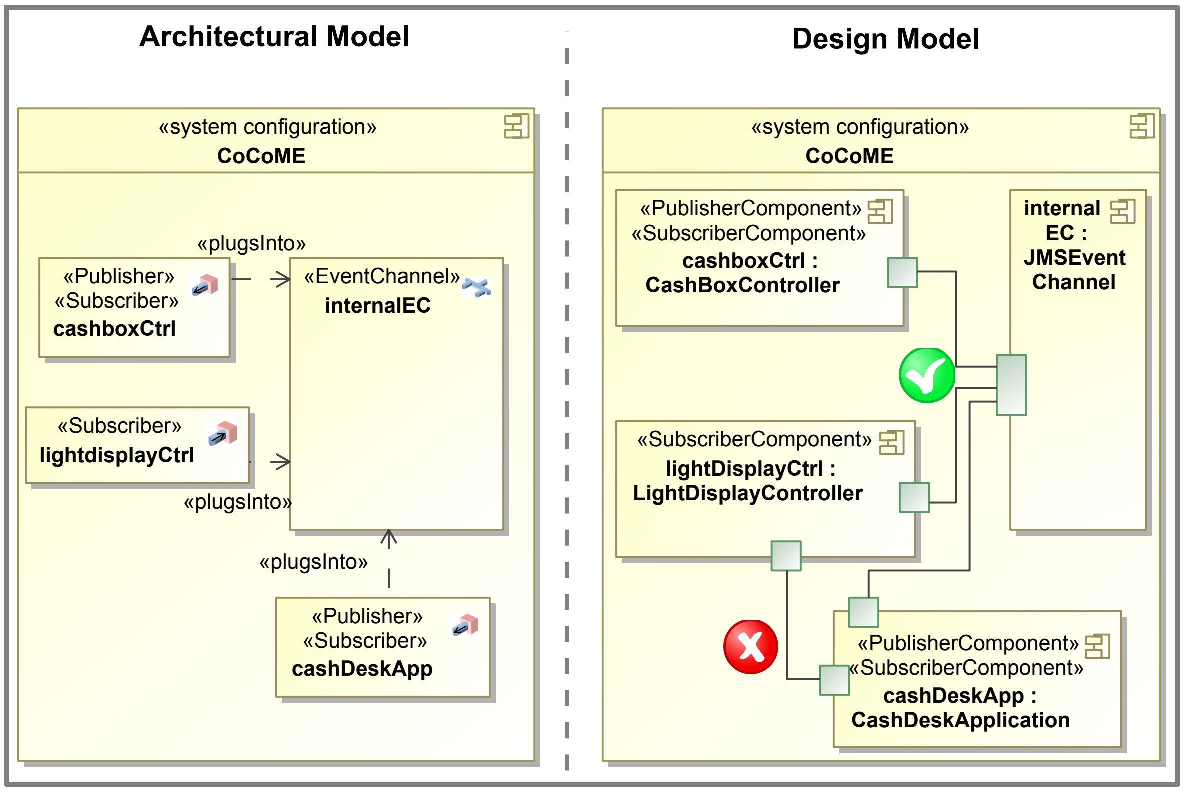


Figure 4.10: Examples of communication conforming to and violating the event-based architecture of the Cash Desk System.

4.3.1. Modification Scenario 1: Adding Components for Cross-Cutting Concerns

In this scenario, the inventory system's functionality is going to be improved by adding *logging* to the inventory system. Logging is the process of recording certain events, actions, or errors that happen during the execution of a system. The recorded information is usually saved to file, a database, or other persistent storage mechanisms. Logging can be helpful during system development, operation and maintenance to understand the behaviour of the system, for example to find sources of erroneous system behaviour. Developers can log, for instance, the values of parameters that are passed to a faulty method to analyse the circumstances leading to occurrences of errors.

The actions that should be logged in this scenario are diverse:

- The activation of GUI elements like opening a new window in the store client software is logged. The name of the activated element and date and time of activation are stored.
- The call of service methods of application layer components is logged. The name of the invoked method, parameter values, and date and time of the invocation are stored.
- The execution of database queries, that are realized and encapsulated in the implementation of the Data::StoreQueryIf interface, is logged. The query string as well as time and date of the query are stored.

Logging is available for many implementation technologies and programming languages "off-the-self", for example the popular log4j logging framework for Java [Gup05]. Most of them can be understood as a component similar to the depicted one in Fig. 4.11. By a functional interface LoggerIf, a specific logger can be created (getLogger(String)). The Logger interface contains methods for setting a logger level, and for writing messages to a log file, a database, etc. By defining a a log level, the developer is able to switch on or off messages of lower relevance, for example to improve execution performance of a system that is rolled-out after testing.

Hence, in every situation that should be logged, the system must be specified to write a message to a logger. This means that the affected components require the interfaces defined by Logger and depend on (at least) this interface. Regarding the layer structure of the inventory system, the question arises to which layer Logger should be assigned.

As a first solution, we consider to add the component to the application layer, depicted in Fig. 4.11. A subpackage for logging is created inside the package Application. All actions that require logging are identified in the design model, and calls to logger.log(...) are added.

An architectural compliance check at this point in time should inform the designer/developer that there is an architectural mismatch. The layer structure is violated. The problem is the logging of database queries in the implementation of StoreQueryIf and other interfaces of the data layer. Since the encapsulated queries should be logged, there is a dependency from the encapsulating component to the required interface Logger located in a layer above.

Unfortunately, moving the logger component to a different layer does not solve the problem. The component and its interfaces are required in all layers but there is no layer that can be accessed from all layers.

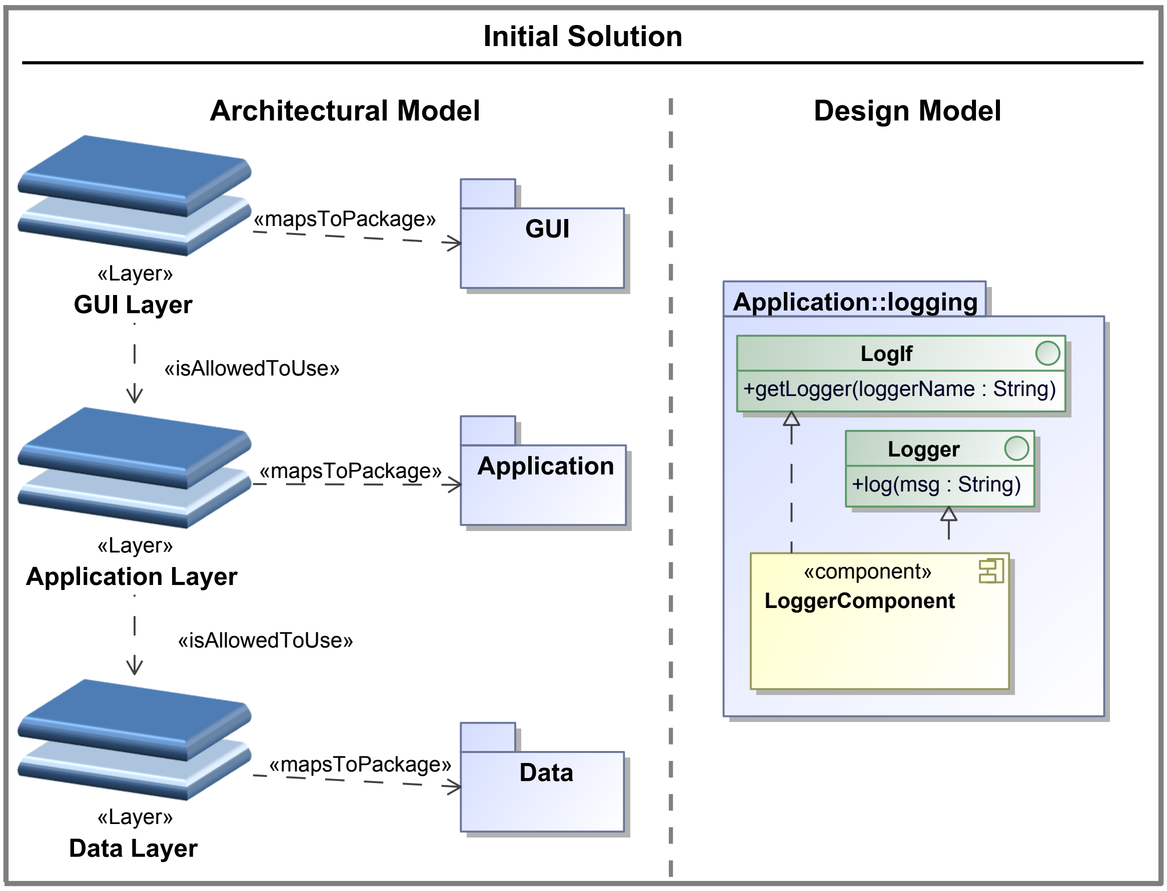


Figure 4.11: Design model of a logger component and incorrect embedding into layers.

The solution is to move the logger component to an additional utility layer that can be accessed from all layers. Logging is a so-called “cross-cutting” concern that is generally required in many components spread over the whole system [CB05]. For such cross-cutting concerns, it is normally not possible to put a realizing component into one layer of a strictly layered structure. Hence, the structure is relaxed by a utility layer as depicted in Fig. 4.12.

The utility layer is positioned “below” the persistence layer and access is granted from all other layers. Consequently, it is architecturally compliant that components from all layers require the interface Logger. After the corresponding changes to the architecture model are applied, the architectural compliance check of the modified design model, containing logger, dependencies towards it, and a refactored package structure, should be successful.

The compliance checks executed in this scenario are described in detail in Sec. 7.3.1. The required architectural rules regarding layers will be defined in Sec. 6.2.1.

4.3.2. Modification Scenario 2: Interface with reference semantics

In this scenario, the stock manager desires a new functionality. He would like to retrieve information about deliveries that are expected to arrive on the current day. A simple information panel shall list those suppliers that are going to deliver during the day.

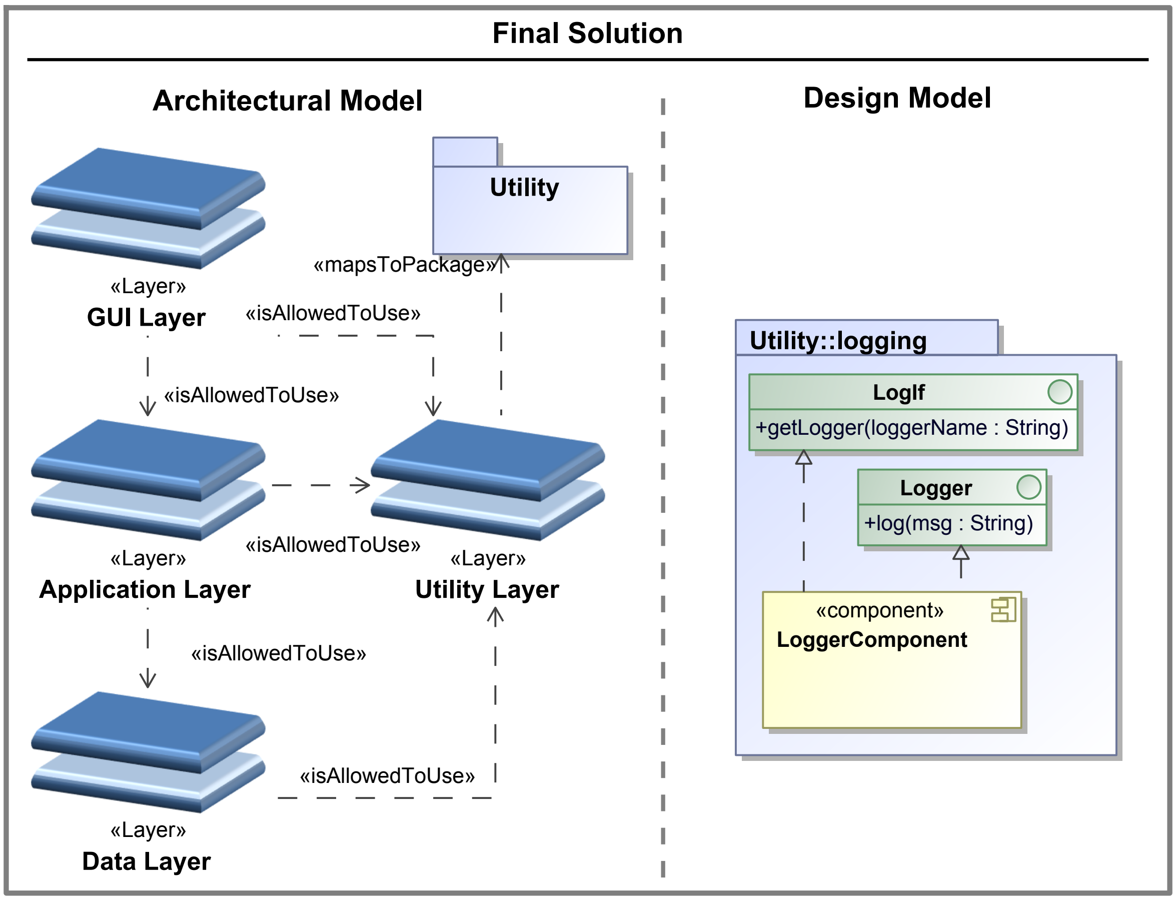


Figure 4.12: Solution for architectural violation in Modification Scenario 1: utility layer for cross-cutting concerns.

The responsible software designer sees that integrating this function is quite simple; basically, it can be realized by a database query selecting from all orders those, which are planned to be shipped today, and joining them with the table of delivering partners. The software designer knows that accessing the data layer directly from the information panel component (being a GUI component) to retrieve the data is not allowed, due to the strictly layered architecture.

Hence, the designer decides to add a new method at the StoreIf interface of the Application::StoreApp component called getSuppliersDeliveringToday. The method uses the query interface of Data::Store to retrieve the required data from the database. The software designer decides to add a corresponding query method to the query interface of the Store component of the data layer. After that, he designs a simple GUI panel. It fills its content during pop-up by calling getSuppliersDeliveringToday, and forwards the Supplier objects to the panel. This solution is depicted in the leftmost section of Fig. 4.13.

The architectural compliance checks executed after these first modelling steps detect architectural mismatches. First of all, there is a dependency between GUI and data layer because instances of ProductSupplier are passed to the new GUI panel. This is a subsequent fault of the second issue; the designer forgot to model the method as a service method as described in

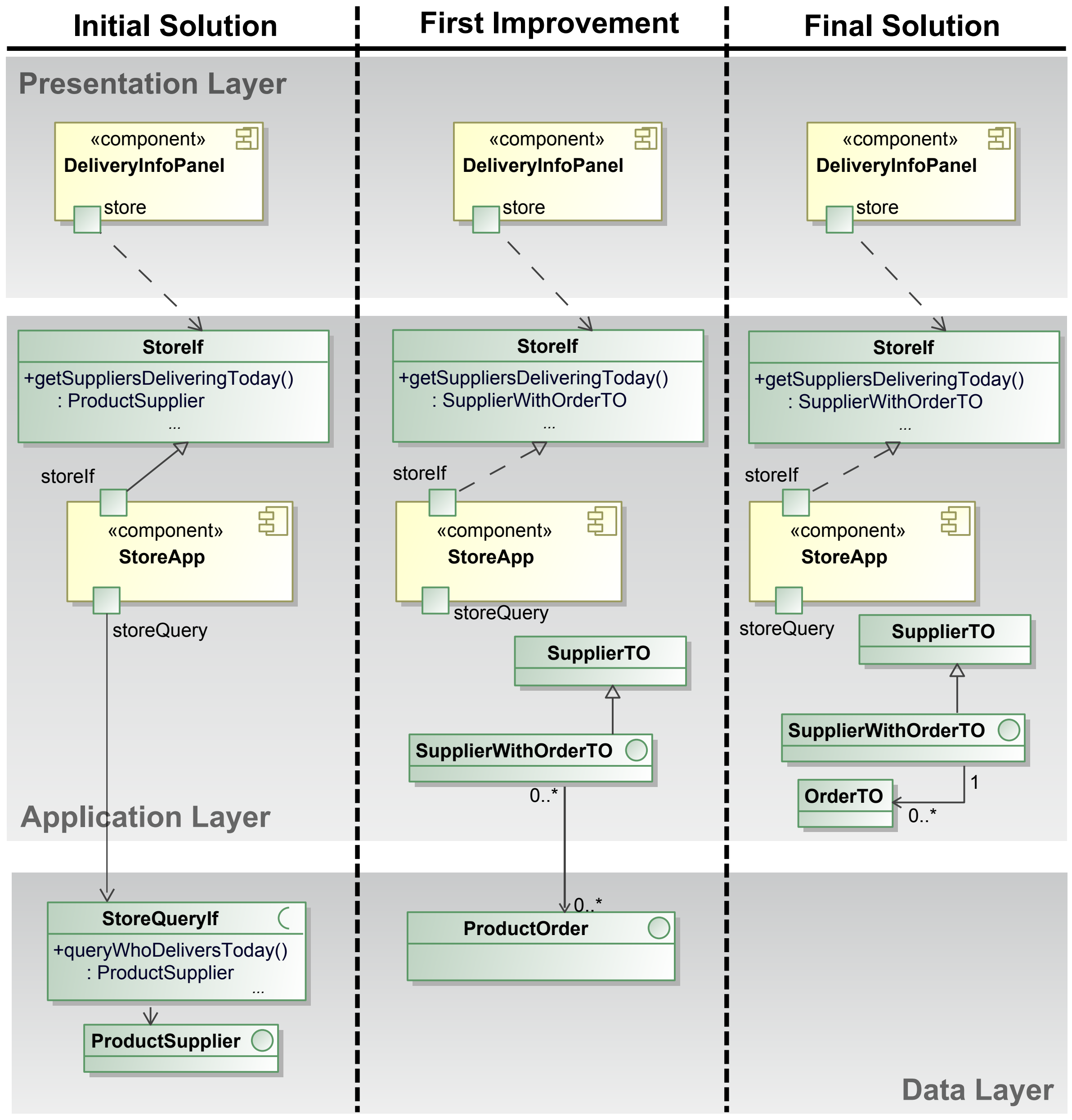


Figure 4.13: Solution for Modification Scenario 2: Designing correct service method and transfer objects

Sec. 4.2.2. The method has to pass transfer objects if it is going to be call from the GUI layer.

The designer seeks advice from the architecture documentation to find out if there are existing transfer object classes he can rely on. Although there exist the interface SupplierTO, he decides that he would like to encapsulate also basic order information in the result of the method, and creates a class SupplierWithOrderTO. This class contains information about a supplier and a list of orders (see middle column of Fig. 4.13).

His first attempt is to keep a reference to an Order object in the new transfer class. Of course, the architecture compliance check shows him that this is not correct because transfer object keeping links to persistent objects lead the concept ad absurdum. Instead, the designer notice his mistake and changes the type of the reference to SupplierTO, an existing transfer object class. After this modification, the compliance checks should pass.

4.3.3. Modification Scenario 3: Direct Communication between Cash Desk Components

In the original design, the cashier at the cash desk is heavily dependent on the correct function of the bar code scanner. The scanner is the only possibility to process goods for sale; a possibility to enter bar codes manually, for example in the case of an inoperable scanner, was not intended. In this scenario, this function is added.

To enter product bar codes manually, the cashier should be able to use the keyboard of the cash box, which is used so far only to enter the amount of money that the cashier receives from a customer, if the latter chooses to pay cash. The designer who has the task to integrate the new functionality notices two things: first, the implementation of the Cash Box component has to be changed. The interpretation of keystrokes now depends on the state of the sale process; before the cashier signals that scanning is finished and payment can start, keystrokes mean that bar codes are entered digit by digit. After that, keystrokes mean digits of amounts of money are entered.

Second, the designer notices that the cash box must now be able to send an event signalling that a product bar code has been entered. He takes a look to the design documentation to get to know the events the bar code scanner of the existing system sends, because that component must be able to do the same thing — to notify the remaining system that a bar code has been entered. The designer finds the interface BarcodeScannedEvent that is appropriate for his purpose.

He re-designs the cash box component a way that it interprets keystrokes depending on the state of the component, and configures that in “bar code mode” the input of a bar code is finished by pressing the “Enter” key; in this case, the cash box sends an BarcodeScannedEvent to the application component. This component can then react to this event as it usually does, i.e. as if the event has been sent by the bar code scanner.

The architectural compliance check, however, notices a violation of the software architecture. The reason for this is the mandatory use of the event channel in the cash desk subsystem of the CoCoME case study. The rules examined in Sec. 4.2.3 prohibit components connected to the same event channel to communicate by the exchange of events directly.

The solution for the mistake in this scenario is quite simple; the designer removes the direct asynchronous call to the application component, as well as the introduced direct connections between cash box and application component. As new and correct solution, he decouples both components and let the cash box publish a BarcodeScannedEvent on the bus when the input of a bar code is finished.

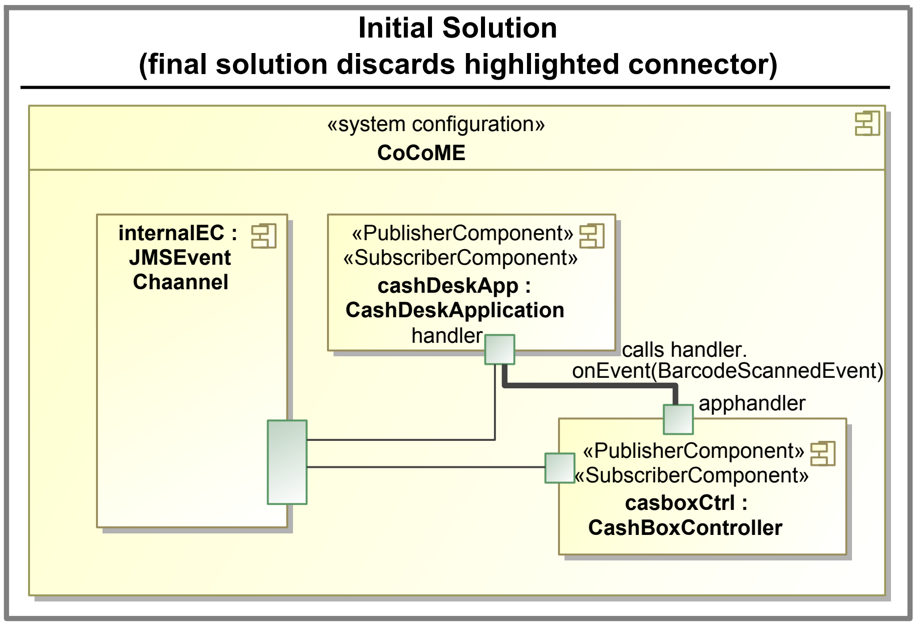

Initial Solution
(final solution discards highlighted connector)
«system configuration»
CoCoME
internalEC : JMSEvent Chaannel
«PublisherComponent»
«SubscriberComponent»
cashDeskApp : CashDeskApplication
handler
calls handler.
onEvent(BarcodeScannedEvent)
apphandler
«PublisherComponent»
«SubscriberComponent»
casboxCtrl : CashBoxController

Figure 4.14: Incorrect solution for modification scenario three: adding direct communication to the Cash Desk System.

4.4. Summary

This chapter has introduced the CoCoME case study serving as an example system for the proposed approach to architectural compliance checking. The case study has characteristics of small real-world software system with regard to complexity, size and architectural aspects.

The discussion so far shows that a flexible approach is required to check architectural compliance in the CoCoME case study. Different architectural aspects, i.e. layered information system, layer interface with copy-semantics, and event-based architecture of the software subsystem running at the cash desks, result in different architectural rules. Hence, it is not sufficient to apply a single approach, e.g. to check layered structures.

Furthermore, it is desirable to check different artefacts of the case study system. The obvious ones are the Java-based implementation as well as the UML design model. But also the content of technology specific artefacts like mapping files for Hibernate can be affected by architectural rules.

Based on these application scenarios, the following chapters will introduce and evaluate the proposed approach. The following chapter will introduce a formal framework to compliance checking which includes the transformation of (architectural) models into a representation which enables us to check rules, especially architectural rules in associated architecture, design, and implementation models.

Chapter 5.

A Formal Framework for Architectural Compliance Checking

Contents

The sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work—that is, correctly to describe phenomena from a reasonably wide area.

John von Neumann

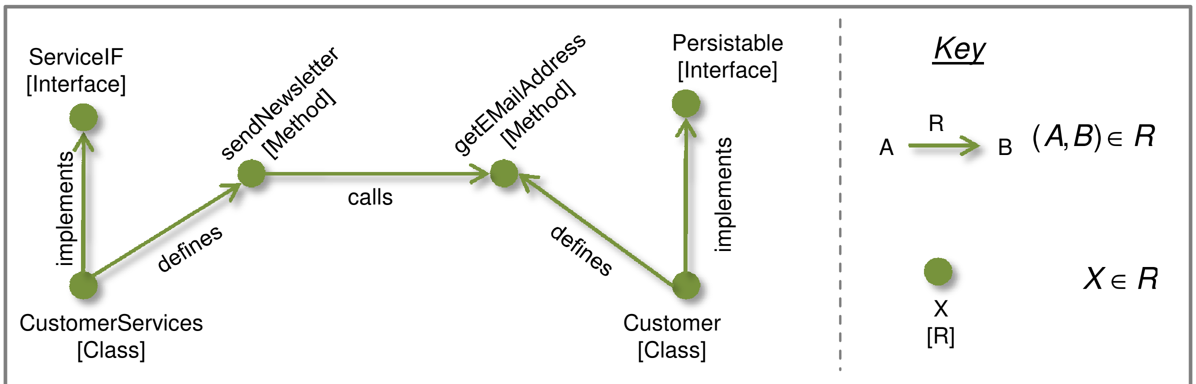
This chapter will explain the conceptual framework developed to support flexible architectural compliance checking in component-based systems. It will describe the concepts and building blocks of the framework in detail. These building blocks are:

- A formal representation of component-based systems and models of component-based systems. The representation can be used for models created in all three system construction stages, i.e. coarse-grained architecture design, detailed design, and implementation.
- Based on the formal representation, a clear classification criteria to distinguish their level of abstraction. This helps us to separate clearly and formally models of the different development phases, and enables us to formally define the compliance relation between them.
- An operationalisation of these concepts to allow tool support for architectural compliance checking. It describes a method to derive systematically the formal representation of models of arbitrary meta models, and to check them for compliance.

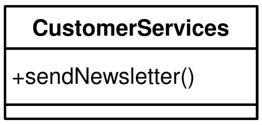
Section 5.1 will give an overview of the building blocks of the framework. It will omit formal definitions as far as possible and will focus on pointing out the main ideas. To prepare the reader for the deepening sections, Sec. 5.2 will introduce some theoretical foundations required to formalize component-based systems, models of systems, and the interrelationships between models. Section 5.3 will address the formal representation of component-based systems and models. In Sec. 5.4, the abstraction level classification for models is introduced and the term of architectural compliance is formally defined. Section 5.5 deals with the operationalisation of the concepts. The chapter is summed up in Sec. 5.6.

5.1. Overview

As mentioned above, a key factor to flexible architectural compliance checking is a formal representation of models of component-based systems and component-based systems themselves. A model that describes a software system is a specification of how the software system must be structured and of how it must behave. For example, a UML composite structure diagram stating that a component C provides an interface I being an Observable implies that a "valid" system contains (at least) C providing I that behaves like an observable (i.e., offering the ability to register observers, notifying observers on state changes, and so on). The model can be



$\text{Universe } U = \{\mathit{ServiceInterface}, \mathit{CustomerServices}, \mathit{sendNewsletter}, \mathit{getEMailAddress}, \mathit{Customer}, \mathit{Persistable}\}$
$\mathit{Interface}^S = \{\mathit{ServiceInterface}, \mathit{Persistable}\}$
$\mathit{Class}^S = \{\mathit{CustomerServices}, \mathit{Customer}\}$
$\mathit{Method}^S = \{\mathit{sendNewsletter}, \mathit{getEMailAddress}\}$
$\mathit{implements}^S = \{(\mathit{CustomerServices}, \mathit{ServiceInterface}), (\mathit{Customer}, \mathit{Persistable})\}$
$\mathit{defines}^S = \{(\mathit{CustomerServices}, \mathit{sendNewsletter}), (\mathit{Customer}, \mathit{getEMailAddress})\}$
$\mathit{calls}^S = \{(\mathit{sendNewsletter}, \mathit{getEMailAddress})\}$



$\varphi = \mathit{class}(\mathit{CustomerServices}) \wedge \mathit{method}(\mathit{sendNewsletter}) \wedge \mathit{defines}(\mathit{CustomerServices}, \mathit{sendNewsletter})$

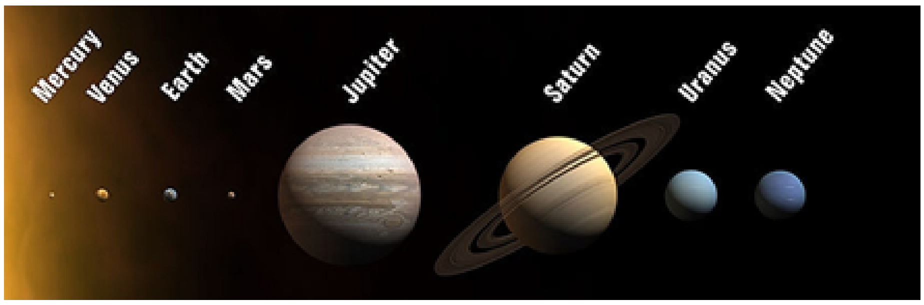
Figure 5.1: A simple system as structure and a model as logical expression.

understood as a constraint that an element of the set of all possible systems has to fulfil to be a conform implementation of the model.

The proposed approach is based upon a formalization of component-based systems as *structures* known from mathematical logic, similar to the approach in [EK03]. Structures consist of a universe of atomic entities and a set of n-ary relations over that universe. Since this is a rather universal definition, software systems can also be represented by structures. Fig. 5.1 depicts a structure representing a software system cut-out. It consists of an entity CustomerService that is an element of the unary relation Class with the meaning that CustomerService is a class. This class provides a method to send newsletters to customers; for this purpose, it calls a getEMailAdress method on the Customer class. The call relation between both methods is represented as a corresponding tuple of the binary relation calls.

The *signature* of a structure defines the relations that are actually available. A signature is a set of relation symbols that are mapped to relations by a concrete structure. Structures that have relations according to a signature τ are called τ-*structures*. τ-structures are used in mathematical logic to define the semantics of first-order logics. A first-order logic formula over τ-structures can contain predicates that relate to the relation symbols from τ and can be evaluated over τ-structures.

Translated to the issue of software system representation, this interrelationship could be used to represent models of software systems. A signature defines the symbols that can be used to describe arbitrary systems, for example the relation symbols Class, Interface, Method, Extension of "Planet" (cut-out)



Intension:
"Objects with true masses below the limiting mass for thermonuclear fusion of deuterium (currently calculated to be 13 Jupiter masses for objects of solar metallicity) that orbit stars or stellar remnants are "planets" (no matter how they formed)."

Figure 5.2: Extension and intension of the concept “Planet”.

implements, defines, and calls. A concrete system, like the system in the upper part of Fig. 5.1, is represented as a structure. The relation symbols are mapped to the specific relations of the structure, for example, Method is mapped to Method^{S}.

If software systems are represented as τ-structures, models could be understood as logic formulae over τ-structures. A system conforms to a model if and only if the formula representing the model is satisfied by the finite structure representing the system[1]. The lower part of Fig. 5.1 shows a simple UML class diagram and the corresponding first-order logic expression that trivially states that the modelled class and its method have to be present in the software system as a conjunction of predicates.

Following this idea, we can roughly distinguish between extensional models and intensional models. Extension and intension are terms that are used in many studies treating the usage of signs, like linguistics, philosophy of language, logic, or mathematics. The extension of a concept, expression or statement consists of the things to which the concept applies. The intension, in contrast, is the set of properties, constraints or ideas that describe the concept. Consider, for example Fig. 5.2, which depicts in the upper part a cut-out of extension of the term “Planet” — namely the planets of the solar system. The complete extension would additionally contain all extra solar planets. Below, the intensional definition of a planet given by the *International Astronomical Union (IAU)* is printed, specifying the properties an interstellar object must have to be considered a planet [2].

The definitions of extension and intension differ in the details from field to field, and the definition of “intensional” is discussed very controversially in the single disciplines by different authors [Car56, Fre92, Tex05].

The approach introduced in [EHK06] by Amnon Eden et al applies the distinction of intensional and extensional statements to first-order logic formulae as representations of models of software systems. An extensional statement, in contrast to intensional statements, is a

[1]in this case, a structure is said to be a model for the statement. To avoid confusion related to the usage of the term “model” in context of MDD, we will use it carefully and only if the actual context is clear.

[2]See http://www.dtm.ciw.edu/boss/definition.html

statement whose set of satisfying structures is closed under adding entities and relation tuples, and removing entities that are not explicitly mentioned by the statement. For example, consider again the class diagram and its statement from Fig. 5.1. Further classes, interfaces, or methods can be added to the depicted structure without losing the property that it satisfies φ. The same holds for entities that are not explicitly referenced in φ, for example Customer. Another property, a statement can have, is that of *locality*. This property expresses that its set of models is closed under adding entities and tuples (in contrast to non-local statements).

Eden uses this classification to classify models of the architecture, design, and implementation level by the degree of abstraction they provide. Eden comes to the conclusion that architectural statements are in general non-local, intensional statements. For example, consider architectural layers that define an hierarchy in a system [BMRS96]. Of course, entities like components can be added to the system in a way that the hierarchy is violated. Thus, the original statement, saying in a first-order logic way that accesses have to be top-down, is violated by an extended structure. The proposed approach will use the classification of Eden and extend it to define the compliance relationship between models of different levels of abstraction. In addition, Eden's framework is in a sense "implemented" by using the general formalization to represent component-based systems.

To make efficient tool support possible, these concepts have to be operationalized. A systematic approach to transform models into structures has to be defined as well as an efficient way to check architectural compliance.

The overall proposed approach is concluded and illustrated in Fig. 5.3. Component-based systems are formalized as τ_{CBSD}-structures conforming to a set of axioms called Φ_{CBSD}. While τ_{CBSD} defines the relation symbols that can be used to describe component-based systems, Φ_{CBSD} defines additional constraints that have to hold for relations in all component-based systems. Descriptions of systems, such as architectural, design, and implementation models are expressed as first-order logic statements. The set of satisfying structures for the corresponding formula is the semantics of a model. An abstraction classification for models of different development phases is introduced based on the general classification of first-order logic statements mentioned above. Based on this separation of different model classes, the term of compliance between models of the three development stages is defined. This definition is operationalised and results in an algorithm to check architectural compliance.

5.2. Foundations

The mathematical foundations of the approach are τ-structures and first-order logics. This section will give important definitions related to those foundations in order to prepare for the formalization of component-based software systems and architectural compliance. The first subsection will define terms related to τ-structures and their function as semantics of first-order logics. The second subsection introduces a classification scheme of first-order logic formulae that will help us to categorize and distinguish architectural, design, and implementation models.

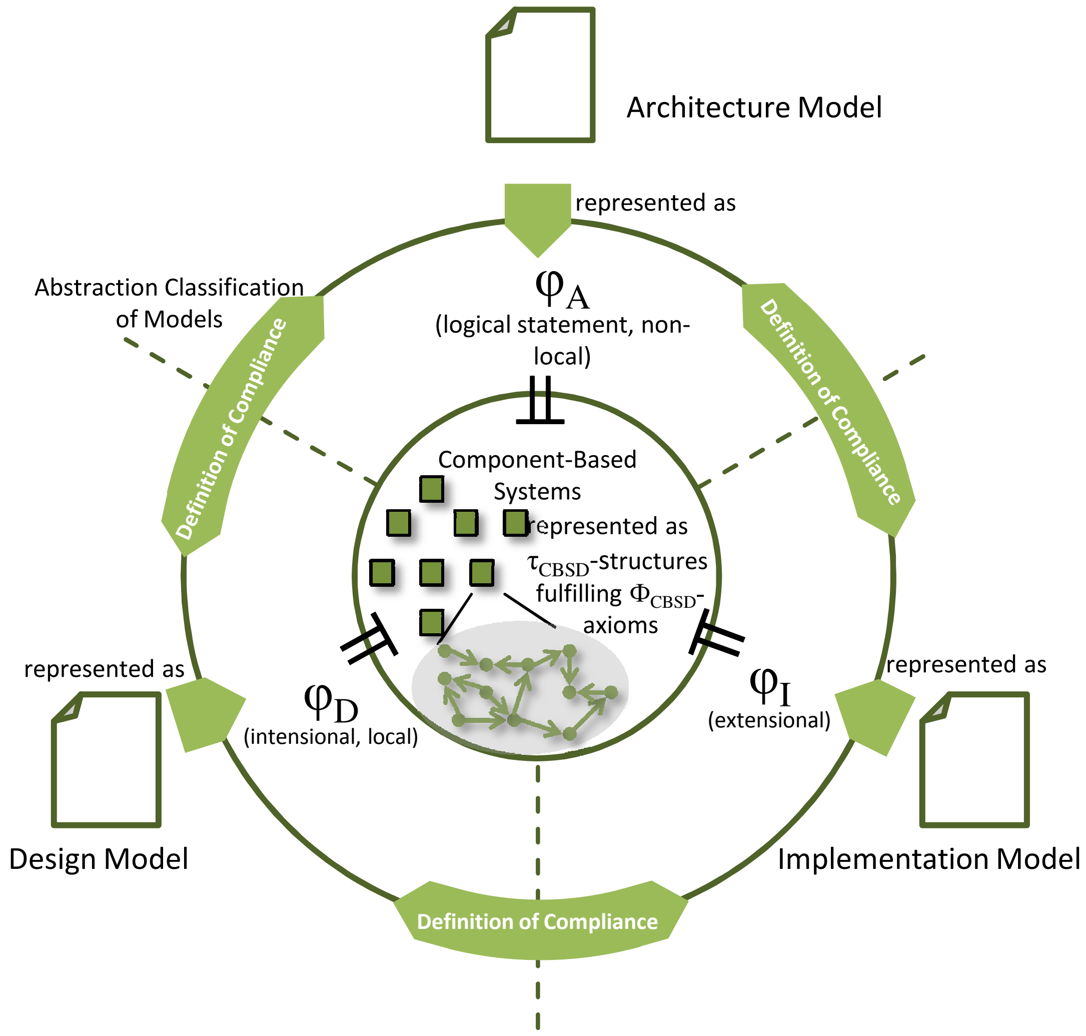


Figure 5.3: Illustration of the framework's components.

5.2.1. Structures

Mathematical *structures* consist of a universe of entities (also called the universe of discourse) and a set of functions and relations over this universe. For example, the set of integers $\mathbb{Z}$ with the functions $+$ and $\cdot$ and the constants 0 and 1 form a structure. The functions and relations that are available in a structure are defined by its signature.

The definition introduced in this section can be found in common textbooks about (first-)order logics as for example [HR04] whereas the notation might vary from book to book.

Definition 1 (Signatures) A *signature* τ is a countable set of function and relation symbols. Every symbol has a finite arity. Function symbols with arity 0 are called *constant symbols*.

In the example above, the signature $\{+, \cdot, 0, 1\}$ provide the functions for integer mathematics. As a further example, a very simple and incomplete signature τ_C for component-based systems could consist of the two unary relations *Component* and *Interface* and the two binary relations *provides* and *requires*.

A signature can be interpreted in the context of a specific structure that maps function symbols and relation symbols to a universe and specific functions and relations.

Definition 2 (τ-structures) A τ-structure $\mathfrak{A} := \left(A, (\,)^{\mathfrak{A}}\right)$ consists of

- a non-empty set of entities A, the *universe* of $\mathfrak{A}$.
- an interpretation function $(\,)^{\mathfrak{A}}$ that maps ever n-ary relation symbol $P \in \tau$ onto a n-ary relation $P^{\mathfrak{A}} \subseteq A^n$, and every n-ary function symbol $f \in \tau$ onto a function $f^{\mathfrak{A}} : A^n \to A$.

$\mathfrak{A}$ is called a *finite structure* if its universe A is finite. In the following, A is implicitly the universe of $\mathfrak{A}$, B the universe of $\mathfrak{B}$, etc., without explicitly denoting the components of the tuples $\mathfrak{A}$, $\mathfrak{B}$, and so on.

Continuing the example from above, a component C providing an interface I and requiring and interface J could be expressed as τ_C-structure $\mathfrak{A}$ like this:

- $A = \{C, I, J\}$
- $Component^{\mathfrak{A}} = \{C\}, Interface^{\mathfrak{A}} = \{I, J\}$
- $provides^{\mathfrak{A}} = \{(C, I)\}, requires^{\mathfrak{A}} = \{(C, J)\}$

In contrast to common definitions from first-order logics, we will allow partial functions and undefined constants to be mapped to function symbols of corresponding arities. The further formalization refers to the approach in [GL90]. In case of an undefined constant for the nullary function symbol c we will write $c^{\mathfrak{A}} = \perp$.

In the following, we will omit the superscript $\mathfrak{A}$ for relations and functions when it is clear which structure is meant and when the relation/function cannot be confused in the respective context with the corresponding symbol.

There are two different ways how a structure can be part of another structure. First, both structures can have the same signature but different universes. Second, they can differ in their assigned signature but have the same universe.

Definition 3 (Substructures and extensions) A structure $\mathfrak{A}$ is called a *substructure of* $\mathfrak{B}$ if all of the following statements hold:

- $\mathfrak{A}$ and $\mathfrak{B}$ have the same signature τ
- $A \subseteq B$, with A being the universe of $\mathfrak{A}$, and B being the universe of $\mathfrak{B}$.
- $R^{\mathfrak{A}} = R^{\mathfrak{B}} \cap A^n$ for every relation symbol $R \in \tau$, n being the arity of R
- $f^{\mathfrak{A}} = f^{\mathfrak{B}} |_{\mathfrak{A}}$ for every function symbol $f \in \tau$.

$\mathfrak{B}$ is also called an *extension* of $\mathfrak{A}$.

For example, the component example from above could be extended by a component D providing the interface C requires. The resulting structure $\mathfrak{A}'$ is an extension of $\mathfrak{A}$:

- $A' = A \cup \{D\}$
- $Component^{\mathfrak{A}'} = Component^{\mathfrak{A}} \cup \{D\}, Interface^{\mathfrak{A}'} = Interface^{\mathfrak{A}}$
- $provides^{\mathfrak{A}'} = provides^{\mathfrak{A}} \cup \{(D, J)\}, requires^{\mathfrak{A}'} = requires^{\mathfrak{A}}$

On the other hand, we could leave the universe of a structure as it is but add or remove functions and relations and get new structures.

Definition 4 (Reductions and expansions) Let σ and τ be signatures with $\sigma \subseteq \tau$, and $\mathfrak{A}$ a τ-structure. The *σ-reduction of* $\mathfrak{A}$ *($\mathfrak{A} \upharpoonright \sigma$)* is the structure that results from removing all relations in $\tau\backslash\sigma$ from $\mathfrak{A}$. More formally this means that $\mathfrak{A} \upharpoonright \sigma := (A', (\,)^{\mathfrak{A}\upharpoonright\sigma})$ is defined as

- $A' = A$
- $(\,)^{\mathfrak{A}\upharpoonright\sigma} = (\,)^{\mathfrak{A}} \mid_{\sigma}$

If $\mathfrak{B}$ is a reduction of $\mathfrak{A}$, then $\mathfrak{A}$ is called a τ-expansion of $\mathfrak{B}$.

For example, we could define a signature $\tau_{C'} := \tau_C \cup \{extends\}$ and create a $\tau_{C'}$-expansion of $\mathfrak{A}''$ with the additional relation $extends^{\mathfrak{A}''} = \{(I, J)\}$.

Structures are used to define the semantics of first-order logics. Formulae, or *statements*, in first-order logics are syntactically build of *τ-terms*. τ terms can be either arbitrary variables from the set of all variables VAR or strings of the form $f(t_1, \ldots, t_n)$ whereas f is a n-ary function symbol from τ and $t_1, \ldots, t_n$ are τ-terms.

τ-formulae are syntactically constructed by applying the known operators (conjunction, disjunction, negation) and quantifiers to predicates that are relation symbols from τ. Terms are used as predicate arguments.

The semantic interpretation of a first-order logic formula assigns a truth value to it that indicates whether the formula holds or not. This depends on the assignment of values to variables of the formula, and the interpretation of function symbols and relation symbols by functions and relations of a specific τ-structure.

Definition 5 (Interpretations, Semantics of first-order logics, minimal structures) Let τ be a signature. A *τ-interpretation* $\mathfrak{I}$ is a tuple $\mathfrak{I} := (\mathfrak{A}, \beta)$ with

- $\mathfrak{A}$ being a τ-structure and
- $\beta : X \rightarrow A$ a function assigning values to variables

whereas VAR is the set of all variables and $X = dom(\beta) \subseteq VAR$.

The interpretation $\mathfrak{I}$ does two things:

- It assigns a value $[\![t]\!]^{\mathfrak{I}} \in A$ to every τ-term t.
- It assigns a value $[\![\varphi]\!]^{\mathfrak{I}} \in \{0, 1\}$ to every τ-formula φ with $free(\varphi) \subseteq dom(\beta)$, $free(\varphi)$ being the set of free variables in φ.

The value assignment is defined inductively according to the syntactical structure of terms and formulae:

For a term t, the value $[\![t]\!]^{\mathfrak{I}}$ is defined by

- for $t = x \in dom(\beta)$: $[\![t]\!]^{\mathfrak{I}} := \beta(x)$
- for $t = f$, $f^{\mathfrak{A}} = \bot$: $[\![t]\!]^{\mathfrak{I}}$ is undefined
- for $t = f(t_1, \dots, t_n)$: if $t_1, \dots, t_n$ are defined, and $f^{\mathfrak{A}}$ is defined at $([\![t_1]\!]^{\mathfrak{I}}, \dots, [\![t_1]\!]^{\mathfrak{I}})$, then $[\![t]\!]^{\mathfrak{I}} := f^{\mathfrak{A}}([\![t_1]\!]^{\mathfrak{I}}, \dots, [\![t_1]\!]^{\mathfrak{I}})$, otherwise undefined.

For atomic formulae φ, the value $[\![\varphi]\!]^{\mathfrak{I}}$ is defined by

- $[\![t_1 = t_2]\!]^{\mathfrak{I}} := \begin{cases} 1 & \text{if } [\![t_1]\!]^{\mathfrak{I}}, [\![t_2]\!]^{\mathfrak{I}} \text{ are defined, and } [\![t_1]\!]^{\mathfrak{I}} = [\![t_2]\!]^{\mathfrak{I}}, \\ 0 & \text{otherwise.} \end{cases}$
- $[\![P(t_1, \dots, t_n)]\!]^{\mathfrak{I}} := \begin{cases} 1 & \text{if } [\![t_1]\!]^{\mathfrak{I}}, \dots, [\![t_n]\!]^{\mathfrak{I}} \text{ are defined, and} \\ & \left([\![t_1]\!]^{\mathfrak{I}}, \dots, [\![t_n]\!]^{\mathfrak{I}}\right) \in P^{\mathfrak{A}}, \\ 0 & \text{otherwise.} \end{cases}$

The values for more complex formulae are defined inductively according to the construction rules for junctors $\neg, \wedge, \vee$ and the quantors $\exists$ and $\forall$ (see [HR04]).

An interpretation $\mathfrak{I} = (\mathfrak{A}, \beta)$ is said to be *a model of the formula* φ, if and only if $[\![\varphi]\!]^{\mathfrak{I}} = 1$. In this case, we write $(\mathfrak{A}, \beta) \models \varphi$ or $\mathfrak{A} \models \varphi[\beta]$. Similar, $\mathfrak{I}$ is a model for a set of τ-formulae Φ, if and only if it is a model for every formula $\varphi \in \Phi$.

If there is a model for a formula φ, φ is called *satisfiable*, otherwise *unsatisfiable*. If it is true in every interpretation, it is called *valid*.

A model s is called *minimal satisfying structure* of a set of formulae Φ if and only if $s \models \Phi$ and there does not exist a substructure s' of s with $s' \models \Phi$.

For example, we could express for the component examples that it should hold that every component must provide at least one interface:

$$\varphi_c = \forall x \exists y : Component(x) \rightarrow provides(x, y)$$

The structures $\mathfrak{A}$, $\mathfrak{A}'$, $\mathfrak{A}''$ as defined above are models for φ_c. But only $\mathfrak{A}'$ and $\mathfrak{A}''$ satisfy the formula φ_i expressing that every interface must be provided by at least one component:

$$\varphi_i = \forall x \exists y : Interface(x) \rightarrow provides(y, x)$$

It is obvious that a τ-formula φ can also evaluated by σ-interpretations, if $\tau \subseteq \sigma$. In this case, the mappings of function and relation symbols from $\sigma \backslash \tau$ are irrelevant with regard to the interpretation of φ.

We will use the notation $\varphi(x_1, \dots, x_k)$ for a formula φ that at most uses $x_1, \dots, x_k$ as free variables. In the following, we will also write $\mathfrak{A} \models \varphi(a_1, \dots, a_k)$ instead of $\mathfrak{A} \models \varphi[\beta]$ to express that $\mathfrak{A}$ satisfies φ with every value assignment β that maps $\beta(x_1) = a_1, \ldots \beta(x_k) = a_k$. We then say "$\mathfrak{A}$ is a model of $\varphi(a_1, \ldots, a_k)$."

A *sentence* is a formula without free variables. In the following, we will also say "$\mathfrak{A}$ is a model of φ" for $(\mathfrak{A},\beta) \models \varphi$ and write $\mathfrak{A} \models \varphi$ if φ is a sentence. Since the function β of an interpretation assigns values to *free* variables, it is irrelevant for sentences.

A set of sentences can be used to define a set of axioms for a class of structures. Each sentence of the set is valid in all structures of that class.

Definition 6 (Model class of logical sentences, axiom systems) Let Φ be a set of τ-sentences. The *model class of* Φ is defined as

$Mod(\Phi) := \{\mathfrak{A} : \mathfrak{A}$ is a structure with $\mathfrak{A} \models \Phi\}$.

We also say, that Φ is the *axiom system for a class* $\mathcal{K}$ if $\mathcal{K} = Mod(\Phi)$.

Formulae can be semantically related, for example, a formula can imply another, or two formulae are equivalent. These relationships can be defined in terms of the sets of models for the related formulae.

Definition 7 (Equivalence of logical formulae) A formula ψ is an *implication* of a set of formulae Φ (written $\Phi \models \psi$), if and only if every model of Φ, that covers the free variables of $\Phi \cup \{\psi\}$, is a model of ψ. Two formulae φ and ψ are *equivalent*, written as $\varphi \equiv \psi$, if $\{\phi\} \models \psi$ and $\{\psi\} \models \phi$.

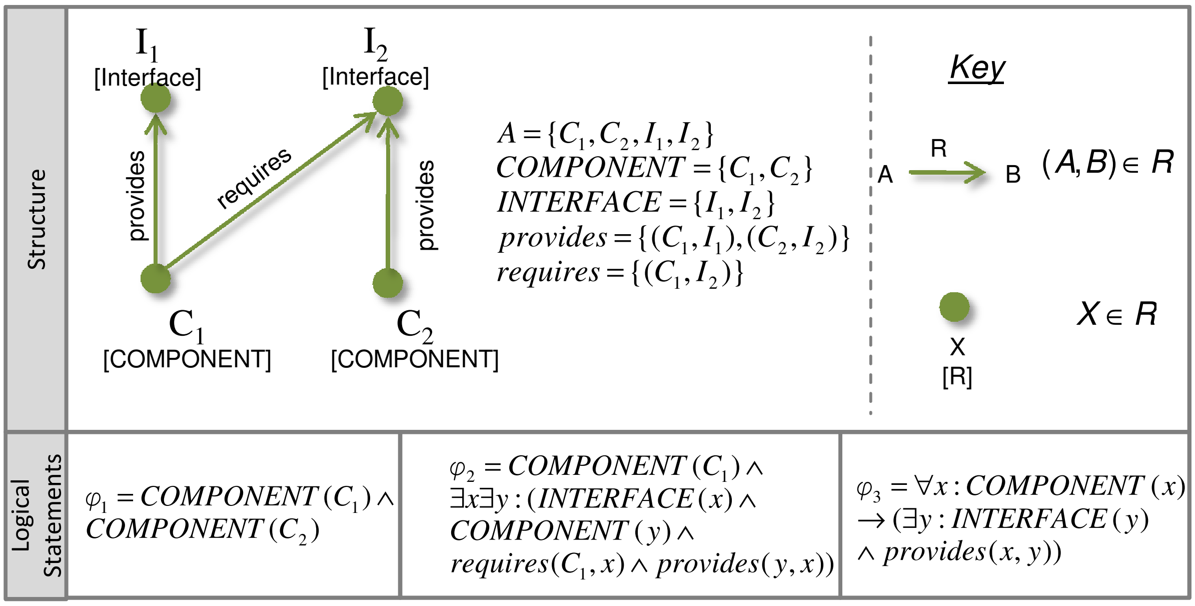
5.2.2. Classification of First-Order Logic Statements

The set of satisfying structures defines the semantics of a formula and enables us to define relationships between formulae like implication and equivalence. Structures can be modified by reducing them to substructures or by extension, i.e. adding or removing elements from the according universe. With the interpretation of structures as software systems, these operations represent changes in a given software system. If a software system satisfies a given description, the question arises if it still does after the modification. In the formalism the question is re-formulated as: does the modified structure still satisfy a given formula φ?

The classification scheme that will be applied distinguishes formulae by the modifications that can be made to models for a formula leading to another model of the same formula. Applied to formulae representing statements about software systems, this scheme distinguishes formally different abstraction levels that descriptions of software systems can provide.

Figure 5.4 depicts a structure in the upper part, and three different statements in the lower part. Obviously, the structure is a model for each of the statements. φ_1 states, that C_1 and C_2 are components. Arbitrary entities could be added to the structure, the modified structure would stay a model of φ_1. Similar, entities could be removed that are not explicitly mentioned in φ_1, such as I_1 or I_2. The removal would not affect the fact that the resulting structure would still be a model of φ_1.

φ_2 is a bit different. It states that C_1 has to depend on some interface provided by another component. The depicted structure satisfies this statement. However, while the addition of arbitrary entities still leads to another model of φ_2, the removal of some of the not explicitly



$A = \{C_1, C_2, I_1, I_2\}$
$COMPONENT = \{C_1, C_2\}$
$INTERFACE = \{I_1, I_2\}$
$provides = \{(C_1, I_1), (C_2, I_2)\}$
$requires = \{(C_1, I_2)\}$

$\varphi_1 = COMPONENT(C_1) \wedge COMPONENT(C_2)$	$\varphi_2 = COMPONENT(C_1) \wedge \exists x \exists y : (INTERFACE(x) \wedge COMPONENT(y) \wedge requires(C_1, x) \wedge provides(y, x))$	$\varphi_3 = \forall x : COMPONENT(x) \to (\exists y : INTERFACE(y) \wedge provides(x, y))$

Figure 5.4: Three statements and a satisfying structure.

mentioned entities will not lead to a model in some cases; the same structure without I_2 and without tuples containing I_2 does not satisfy φ_2.

For φ_3, neither adding nor removing entities is "safe". φ_3 states that every component has to provide at least one interface. Obviously, removing I_1 or I_2 results in a structure that is not a model for φ_3. Additions as well can lead to such a structure, for example adding a single entity C_3 being a component, would create a structure that does not satisfy φ_3.

φ_1, φ_2, and φ_3 are instances of three different classes of statements, called implementation statements (φ_1), tactical statements (φ_2), and strategic statements (φ_3) [EHK06]. In the following, we will define them formally.

First, we would like to distinguish different extensions of a structure $\mathfrak{A}$. If $\mathfrak{A}$ is extended by adding a tuple of existing entities to a relation, the resulting structure is called a *modifying extension of* $\mathfrak{A}$. An extension is also modifying, if an added tuple refers to existing and added entities but the relation as such is interpreted as changing the existing entity because the relation symbol stands for a very "close" relationship. Consider, for example, adding a parameter to a method, what could be modelled as a binary relation between two entities, would rather be understood as modifying the method than as extending it.

More formally, this is stated in the following definition:

Definition 8 (Modifying and additive extensions of structures) Let the τ-structure $\mathfrak{B}$ be an extension of a given τ-structure $\mathfrak{A}$ with $B = A \dot{\cup} \{e_1, \ldots, e_k\}$. $\mathfrak{B}$ is called a *modifying extension of* $\mathfrak{A}$ *w.r.t* $\tau_c \subseteq \tau$ if

- there is $(x_1, \ldots, x_n) \in R^{\mathfrak{B}}$ with $x_i \in A$ for $i = 1, \ldots, n$ and $(x_1, \ldots, x_n) \notin R^{\mathfrak{A}}$, or
- there is $(x_1, \ldots, x_n) \in R^{\mathfrak{B}}$, $R \in \tau_C$, $x_1 \in A$, and $x_i = e_j$ for at least one $i \in \{2, \ldots, n\}$.

Otherwise, $\mathfrak{B}$ is called an *additive extension of* $\mathfrak{A}$ *w.r.t* τ_C.

Second, we will define the partial structure of a structure that results from a "clean" removal of entities. This means that also tuples are removed that contain an entity that is removed. Furthermore, constant symbols may not longer be mapped to deleted entities, which neither can be in the domain of functions.

Definition 9 (Partial structure) For a given structure $\mathfrak{A}$, the structure $\mathfrak{A}\backslash\{e_1, \dots, e_k\} = \mathfrak{B}$ is defined as

- $B = A\backslash\{e_1, \dots, e_k\}$
- for all relation symbols R: $R^{\mathfrak{B}} := R^{\mathfrak{A}} \cap B^n$
- for all constant symbols c: $c^{\mathfrak{B}} = c^{\mathfrak{A}}$ if $c^{\mathfrak{A}} \neq e_i$, for $i = 1, \dots, k$, otherwise $c^{\mathfrak{B}} = \perp$
- for all other function symbols f: $f^{\mathfrak{B}} = f'^{\mathfrak{A}} |_B$, whereas

$$f'^{\mathfrak{A}}(x_1, \dots, x_n) := \begin{cases} f^{\mathfrak{A}}(x_1, \dots, x_n) & \text{if } x_i \text{ and } e_j \text{ mutually different} \\ & \text{for } i = 1, \dots, n \text{ and } j = 1, \dots, k \\ (x_1, \dots, x_n) \notin dom(f'^{\mathfrak{A}}) & \text{otherwise.} \end{cases}$$

As stated above, only entities can be considered for removal from models for a formula φ that are not explicitly mentioned in φ. Of course entities cannot be used directly in formulae because the elementary terms to construct formulae are only variables and symbols from the signature that the formula is defined for. Thus, an explicit mention of entities in a formula is defined as follows:

Definition 10 (Explicitly mentioned entities) Given are a τ-structure $\mathfrak{A}$ and a τ-formula φ. An entity $e \in A$ is *explicitly mentioned in* φ if φ contains a term $t = c$ whereas c is a constant symbol from τ and $c^{\mathfrak{A}} = e$.

With these definitions, we are able to define the category of extensional first-order logic statements.

Definition 11 (Extensional and intensional formulae) A first-order logic τ-formula φ is called *extensional (w.r.t to $\tau_C \subseteq \tau$)* if both of the two following statements hold for every τ-structure $\mathfrak{A}$ with $\mathfrak{A} \models \varphi$:

- for every subset $A' \subseteq A$ of entities that are not explicitly mentioned in φ is $\mathfrak{A}\backslash A' \models \varphi$.
- for every additive extension (w.r.t. τ_C) $\mathfrak{B}$ of $\mathfrak{A}$ is $\mathfrak{B} \models \varphi$.

φ is called *intensional (w.r.t. τ_C* if and only if it is not extensional.

In the example depicted in Fig. 5.4 only φ_1 is an extensional statement. The depicted structure is an example which does not satisfy the constraints of the definition above regarding φ_2 and φ_3. Hence, both statements are intensional.

Another classification criterion for first-order logic statements is locality.

Definition 12 (Local and non-local formulae) A first-order logic τ-formula φ is called *local (w.r.t. $\tau_C \subseteq \tau$)* if for every τ-structure $\mathfrak{A}$ with $\mathfrak{A} \models \varphi$ holds: every additive extension (w.r.t. τ_C) $\mathfrak{B}$ of $\mathfrak{A}$ is a model of φ ($\mathfrak{B} \models \varphi$). A formula φ is called *non-local (w.r.t. τ_C)* if and only if it is not local.

It is obvious that every extensional statement is also local, since the constraints for extensional statements include those for local ones. For the same reason, every non-local statement is intensional but not vice versa — intensional statements can either be local or non-local. φ_2 (see. Fig. 5.4), for example, is intensional and local, φ_3 is intensional and non-local. In the context of software systems, we adapt the nomenclature of Eden [REF] who refers to implementation statements (extensional), tactical statements (intensional, local), and strategic statements (non-local).

5.3. Formal Representation of Component-Based Systems

As introduced in Sec. 5.1, the proposed approach is based upon a representation of component-based systems as structures of a common signature τ_{CBSD}. τ_{CBSD} defines the symbols required to formalize all relevant aspects of component-based systems, and to formulate logical statements about them. The relation symbols of τ_{CBSD} define the available types of entities such as components and interfaces, and the types of relationships between them, such as "provides" or "requires". An axiom system is defined describing restrictions that are valid for component-based-systems in general; for example, it states that a connection between components can only be specified if the connected ports are type-compatible. The axiom system for component-based systems is called Φ_{CBSD} in the following.

This section will explain τ_{CBSD} and Φ_{CBSD} in detail. The next subsection will give a quick overview of τ_{CBSD} and the general form of axioms of Φ_{CBSD}. After that, the subsequent sections will dive into the details of the single subsets of τ_{CBSD} one by one.

5.3.1. Overview of the formalization of Component-Based Systems

Structures representing component-based systems are characterized by τ_{CBSD} on the one hand, on the other hand by the axiom system Φ_{CBSD}. While τ_{CBSD} defines entity types and relationships between them, Φ_{CBSD} defines constraints that have to hold in component-based systems in general. More formally, we can define:

Definition 13 (Component-Based Systems) A component-based system $\mathfrak{A}$ is an element of $\mathcal{K}_{CBSD} := Mod(\Phi_{CBSD})$ whereas for the universe A of $\mathfrak{A}$ holds: $A \subset \mathbb{N}$.

Please notice, that the definition of model classes as used above implies that also τ'-structures with $\tau_{CBSD} \subseteq \tau'$ can be component-based systems as long as the axioms of Φ_{CBSD} are valid for them.

The reader familiar with UML and model-driven software development in general will in the following notice some similarities between the concepts and terms used for τ_{CBSD} and the meta model of UML [Obj10b]. On the one hand, UML as a general purpose modelling language covers also component-based systems and, thus, has language elements to cover components and related concepts. As a standardized language supported and developed by experienced software developers and designers, these concepts are elaborated very well. Consequently, some of them inspired and influenced the view on component-based systems presented here.

The main differences between UML and the signature for component-based systems τ_{CBSD} along with the axiom system Φ_{CBSD} are the particular purposes of use. UML is a modelling language which can be used to describe systems at different abstraction levels, for example component-based systems. In contrast, τ_{CBSD} and Φ_{CBSD} provide a formal, mathematical construct to define the semantics of models describing component-based systems.

Furthermore, the meaning of the concept of a component and related concepts and their distinction from other concepts remain very unclear in UML. Most of the properties of a component also apply to classes in UML, a fact that manifests itself as a common super meta class “Classifier” that plays a central part in this part of the meta model. Classifiers, and hence classes as well as components, can be connected by associations, can have properties, can be specialized, and can be composite structures. Thus, the differences between object-oriented and component-based concepts are blurred and overloaded.

The separation of concepts supposed here is clear and stringent. A component is a complex and composite, autarkic type that defines by interfaces (being collections of method signatures) which functionalities it requires and provides. Classes are used as component-local types only. Due to the fact that the goal of this approach is more focused on a single kind of systems, concepts can be defined more lightweight than in a general-purpose-language like UML.

However, defining τ_{CBSD} and Φ_{CBSD} is similar to creating a meta model for a modelling language for component-based systems. Despite the different usage purposes mentioned above, we will use class diagrams in addition to the pure mathematical notations. They provide a convenient illustration mean to a quick understanding of which relation symbols and axioms are contained in τ_{CBSD} and Φ_{CBSD}, respectively. The mapping between class diagrams and both structures will be explained in the following subsections.

τ_{CBSD} from a Bird’s Eye Perspective

τ_{CBSD} can be divided into six disjoint subsets, each covering a different aspect of component-based systems. These subsets are:

$$\tau_{CBSD} = \tau_{gen} \dot\cup \tau_{type} \dot\cup \tau_{oos} \dot\cup \tau_{cs} \dot\cup \tau_{bhv} \dot\cup \tau_{id}$$

All relation symbols of τ_{CBSD} are either unary or binary. Unary relation symbols stand for entity types while binary relation symbols model relationships between entities. The only constant symbols are contained in τ_{id} to refer to entities in formulae. No further function symbols are defined apart from that.

τ_{gen} consists of general relation symbols that are used and refined throughout the whole signature. For example, a relation symbol “NamedElement” is used as a type for entities that are endowed with a name, such as components and interfaces, or a relation symbol “Package” which provides a namespace mechanism.

τ_{type} contains symbols for the main constructs to describe component-based systems on the type level. Those are, for example, components and interfaces. They will be covered in Sec. 5.3.2.

In τ_{oos}, the object-oriented aspects of component-based systems will be covered. The contained relation symbols model classes, inheritance, and so forth. They are described in Sec. 5.3.3.

The relation symbols in τ_{cs} deal with the inner structure of components that is defined by ports, parts, and connectors. The structure of an overall system is very similar to the structure of single components, thus most of the constructs are reused to define system configurations. They will be explained in Sec. 5.3.4.

τ_{bhv} is the subset of relation symbols relevant for the behaviour description of systems and components. For example, communication issues like method invocation or asynchronous messaging, instance creation, etc. are reflected as relation symbols. They are explained in Sec. 5.3.5.

τ_{id} is a countable infinite subset of constant symbols that identify entities from a structure's universe which is the set of integers. It is defined as

$$\tau_{id} := \{e_i \mid i \in \mathbb{N}\}.$$

An interpretation $[\![e_i]\!]^{\mathfrak{A}} = i$ pointing to an entity of the universe of $\mathfrak{A}$ means that this entity has been allocated the identifier e_i. $[\![e_i]\!]^{\mathfrak{A}} = \perp$ indicates that no such entity exists. With these constant symbols, we are able to let formulae refer to entities by their identifier.

As already mentioned, class diagrams will be used in the following sections to illustrate the subsets of τ_{CBSD}. Figure 5.5 depicts the general mapping from class diagrams to relation symbols. For every class Class_A, there is a unary relation symbol "Class_A". Attributes of classes are mapped to binary relation symbols (see attr), whereas the first component of a tuple refers to the class owning the attribute, and the second component to the attribute itself. The same mapping is applied to association ends as far as they are named, e.g. rel in Fig. 5.5. Of course, the specific interpretations of relation symbols in form of relations in specific structures underlie additional constraints — for example, "rel" may only relate an entity e with an entity f if $e \in Class\_A$ and $f \in Class\_B$. These constraints and others are part of the axiom system Φ_{CBSD}, which is described in the following section.

The Axiom System Φ_{CBSD}

The axiom system $\Phi_{CBSD} = \{CBSD_i \mid i = 0, 1, \ldots\}$ contains τ_{CBSD}-sentences expressing constraints that have to hold in component-based systems in general. Many of these constraints are specific to a certain type of entity or a specific relationship. For example, we are going to express that a class can only inherit another class if they are encapsulated by the same component. Contrarily, there are some general classes of constraints that are represented as sentences of similar shape for different relation symbols:

- *Constraints defining the domain of unary relation symbols.* Unary relation symbols represent types of entities. To express subtyping, the relationship between subtype and

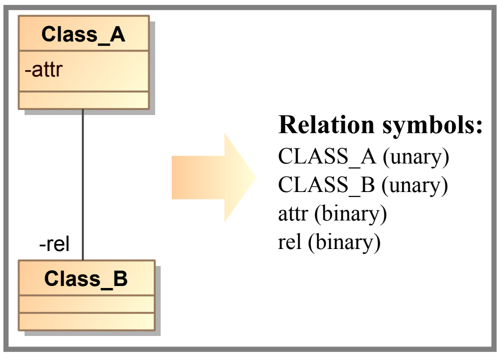


Figure 5.5: Mapping of meta model elements to relation symbols.

supertype has to be axiomatically be defined for the corresponding relation symbols. For example, every component is also a named entity. This is formalized as sentence that states: e being a component implies that e is also a named element. This scheme can be generalized.

- *Constraints defining the domain of binary relation symbols.* These constraints restrict the domain of relationships between entities to certain types of entities. For example, it must be axiomatically stated that *provides* is a relation between components and interfaces only.

- *Constraints defining the multiplicity an entity may appear as a component of a relation tuple*. These constraints, for example, state that a single entity may participate in a relation exactly once. For instance, a component c must have exactly one name and, thus, there must be exactly one tuple of the form $(c, MyComponent) \in name$ for a fixed c.

Constraints of the first kind are in general defined as follows:

$$CBSD_i = \forall x : T(x) \rightarrow T'(X) \tag{5.1}$$

whereas the unary relation T denotes a type that is subtype of a type denoted by T', meaning that every entity being of type T is also of type T'. It indicates that $[\![T]\!]^{\mathfrak{A}} \subseteq [\![T']\!]^{\mathfrak{A}}$ for every $\mathfrak{A} \in \mathcal{K}_{CBSD}$. Furthermore, every entity in a component-based system can only be in one type hierarchy — for example, an interface cannot be a component at the same time. Since the set of unary relation symbols representing types is finite, it can be stated as first-order logics statement:

$$CBSD_i = \forall x : T(x) \rightarrow \neg T_1(x) \wedge \neg T_2(x) \wedge \ldots \wedge \neg T_k(x) \tag{5.2}$$

with $T_1, \ldots, T_k$ being all unary relation symbols representing types that are neither subtypes nor supertypes of the type represented by T.

Constraints that define the domain of a relation R to certain types T_1 and T_2 are defined as

$$CBSD_i = \forall x \forall y : R(x,y) \rightarrow T_1(x) \wedge T_2(y) \tag{5.3}$$

These axioms ensure for given $R, T_1,$ and T_2, that in every component-based system $\mathfrak{A}$ holds that $[\![R]\!]^{\mathfrak{A}} \subseteq [\![T_1]\!]^{\mathfrak{A}} \times [\![T_2]\!]^{\mathfrak{A}} \subseteq A \times A$.[3]

For constraints limiting the multiplicity of an entity as a fixed component of relation tuples, four cases are distinguished.

1. The multiplicity can range between 0 and an arbitrary number.
2. The multiplicity can range between 1 and an arbitrary number.
3. The multiplicity can range between 0 and 1.
4. The multiplicity must be exactly one.

The first case does not require any constraints. It means that, for a given entity x and a given relation *rel*, that there might be an arbitrary number (including zero) of entities y with $(x, y) \in$ *rel*. In the second case, zero is not allowed, thus, there must be at least one y. Hence, the constraints in this case are defined as, assuming that *rel* is a binary relation between entities of type T and T':

$$CBSD_i = \forall x : T(x) \rightarrow \exists y : rel(x,y) \tag{5.4}$$
$$CBSD_j = \forall y : T'(y) \rightarrow \exists x : rel(x,y) \tag{5.5}$$

The first formula states that, for a fixed element at the first component, there is at least one tuple. The second formula, as in the following pairs of axioms, states the same but for a fixed entity at the second component.

The third case, the multiplicity ranging between 0 and 1, is defined by the constraints

$$CBSD_i = \forall x : T(x) \rightarrow ((\forall y : \neg rel(x,y)) \vee (\exists! y : rel(x,y))) \tag{5.6}$$
$$CBSD_j = \forall y : T'(y) \rightarrow ((\forall x : \neg rel(x,y)) \vee (\exists! x : rel(x,y))) \tag{5.7}$$

In the case of a multiplicity of exactly one, we can write

$$CBSD_i = \forall x : T(x) \rightarrow \exists! y : rel(x,y) \tag{5.8}$$
$$CBSD_j = \forall y : T'(y) \rightarrow \exists! x : rel(x,y) \tag{5.9}$$

Table 5.1 shows how class diagrams as illustrations of τ_{CBSD} can be mapped to the axioms of the before mentioned shapes. Generalization obviously models the subtype relationship

[3]This is abbreviated as $R \subseteq T_1 \times T_2$.

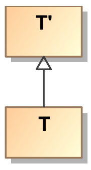
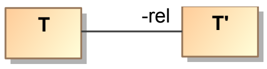
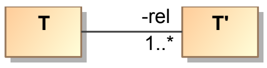
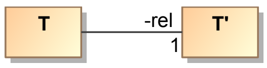
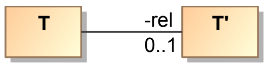
	Class diagram	**Axioms**
Domain restriction for unary relation symbols (subtyping)		see axioms 6.1 and 6.2
Domain definition for binary relation symbols		see axiom 6.3
Multiplicity "at least one"		see axioms 6.4 and 6.5
Multiplicity "exactly one"		see axioms 6.6 and 6.7
Multiplicity "zero or one"		see axioms 6.8 and 6.9

Table 5.1: Class diagram notation for τ_{CBSD} and corresponding axioms of Φ_{CBSD}.

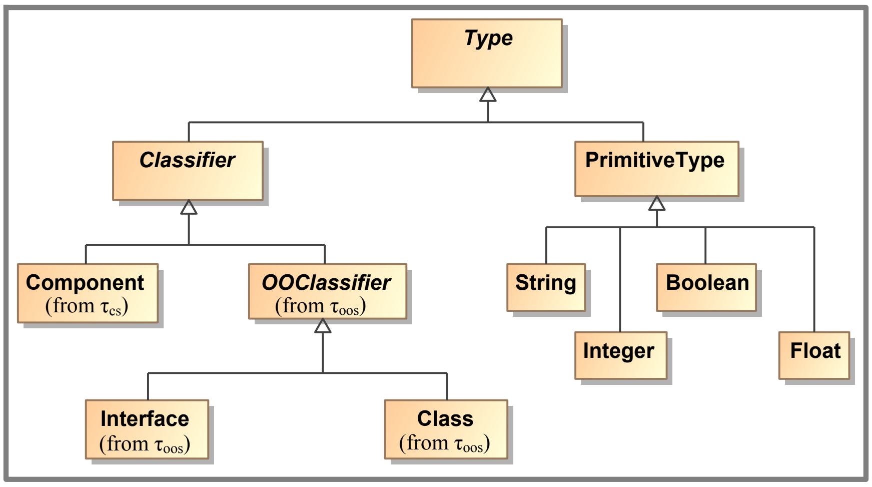

Type
Classifier
PrimitiveType
Component (from τ_{cs})
OOClassifier (from τ_{oos})
String
Boolean
Integer
Float
Interface (from τ_{oos})
Class (from τ_{oos})

Figure 5.6: Hierarchy of type relation symbols.

between entity types. Meta classes connected by meta associations define the domain of the binary relation symbol(s) represented by the corresponding association ends. Their multiplicity determines which of the multiplicity constraints are contained in Φ_{CBSD}.

In the following sections describing τ_{CBSD}, these general constraints defined for almost every symbol of τ_{CBSD} will not be mentioned explicitly. They can be found in App. A. Only entity type specific or relationship specific constraints will be explained explicitly in this chapter.

5.3.2. Types and Typed Elements(τ_{type})

The subset τ_{type} of τ_{CBSD} defines relation symbols to formalize different kinds of *types* used in component-based software development. A type can be understood as a constraint over the data values an element can represent. For example, “integer” in a programming language constrain the possible values an element of this type can have to a finite subset of integers. Types are modelled in τ_{type} as relation symbol $Type \subseteq NamedElement$ and subsetted by a set of more specific kind of types, as depicted in Fig. 5.6.

Beside primitive types (integer, strings, boolean, etc.), the main types used in component-based systems are components, interfaces, and classes. All of them define complex types that have structure and behaviour. Components, captured by $Component \subseteq Classifier$ (see Fig. 5.6), represent modular parts of a system at the type level, and encapsulate, or hide, their specific implementation from the rest of the system or the environment. The functionality a component provides or requires is defined by interfaces. A component requiring an interface I must be “wired” to a component that provides I in order to realize its own functionality. An interface declares the structure of a type by declaring signatures of methods and member variables. An interface itself cannot be instantiated but needs to be implemented by classes. Classes and interfaces are summarized by the relation symbol $OOClassifier \subseteq Classifier$ to group these concepts from object-orientation.

Fig. 5.7 depicts which elements can actually be typed. Members of classes or interfaces can be typed as well as parameters of methods and local variables in method bodies. Parts are a superset

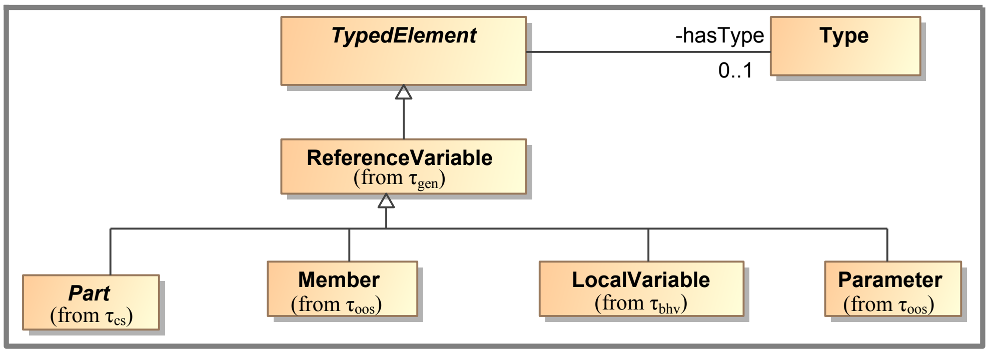


Figure 5.7: Entity types that can be assigned types from $\tau type$.

of different kinds of constituent parts that components are made of. All of these entity types as subsets of *TypedElement* can participate in the relation $hasType \subseteq TypedElement \times Type$ that reflects the typing.

Main Type Constructs: Components and Interfaces

Components and interfaces are the main type constructs in component-based systems. They rely heavily on concepts known from object-orientation like classes, inheritance, object references, etc. Interfaces themselves are known in object-orientation as well. τ_{oos} contains relation symbols that capture the structural aspects of object-oriented specifications. Figure 5.8 depicts τ_{oos} and the most important connections to other relation symbols of τ_{CBSD}. These are in particular connections to *Component*.

$Component \subseteq Type$ represents the set of components, while interfaces are formalized as elements of the unary relation symbol $Interface \subseteq Type$ (see Fig. 5.8). The relation symbols $providesInterface \subseteq Component \times Interface$ and $requiresInterface \subseteq Component \times Interface$ connects each component with the interfaces it provides or requires. The classes a component encapsulates are reflected by the relation symbol
$containsClass \subseteq Component \times Class$.

Components and interfaces can be grouped by packages that are modelled by $Package \subseteq NamedElement$. Packages provide namespaces, in which names of types are unique. Every component and every interface must be contained in exactly one package. The relation symbol $containsPkg \subseteq Package \times Package$ allows us to compose hierarchical package trees. $(p_1, p_2) \in containsPkg$ indicates that p_2 is a subpackage of p_1. $containsPkg^+$ denotes the transitive closure of $containsPkg$; $(p_1, p_2) \in containsPkg^+$ indicates that p_1 contains p_2, possibly indirectly.

It is worth noting for readers familiar with object-oriented concepts and languages, that classes are not considered as elements grouped in packages and accessible from anywhere in the system. Often also classes are entities that are provided by components directly, e.g. if they reflect simple data structures with only getter and setter methods; in this case, the interface and implementation of a class is integrated into a single unit, and not strictly separated. Since this is not allowed in τ_{CBSD}, the most straight-forward conversion of such “data interfaces” would be to assume a standard implementation encapsulated in the component, which provides simple getter and setter implementations. Alternatively, τ_{CBSD} could be easily extended to integrate

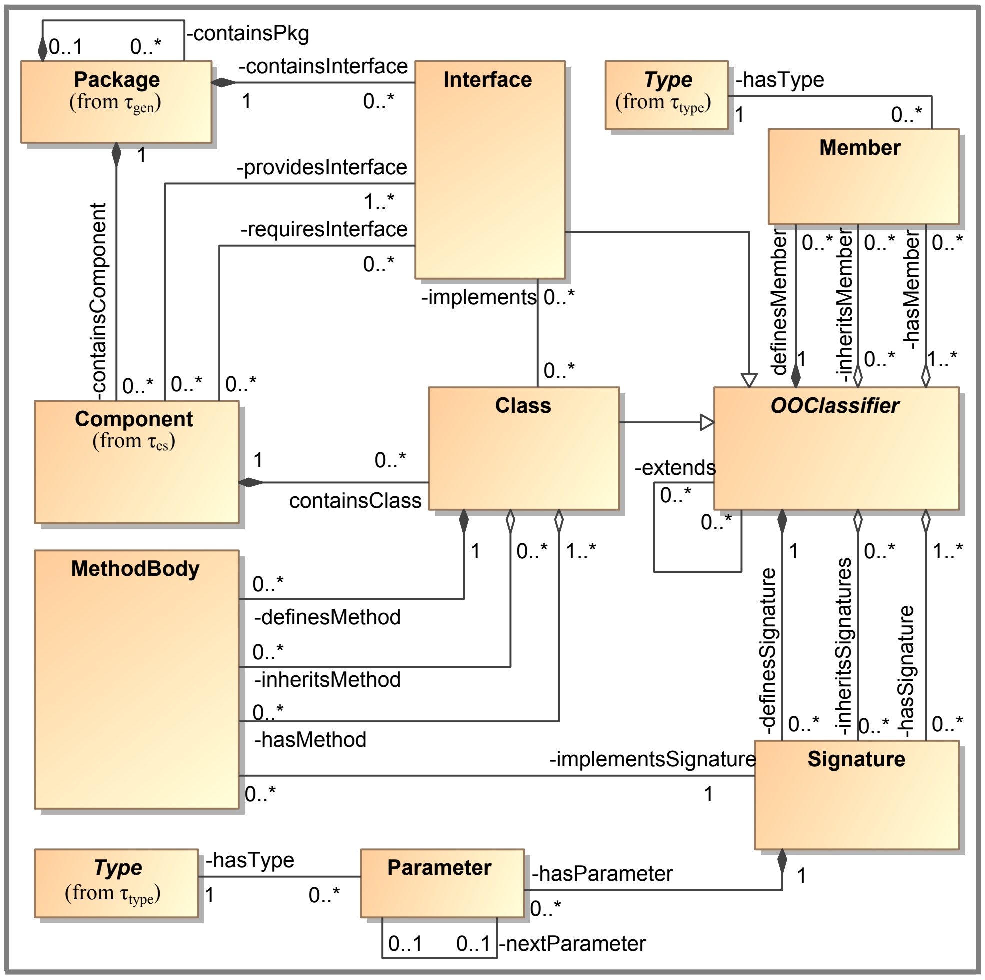


Figure 5.8: Relation symbols of τ_{oos} that model the structure of classes and interfaces, and relation symbols that connect them to other subsets of τ_{CBSD}.

complex data types as specialization of *PrimitiveType*, such as records or sequences.

Running Example

Figure 5.9 depicts an example in UML notation illustrating the relation symbols introduced so far. Notice, that the diagram is annotated with «System diagram». This indicates that the diagram depicts a *system* in the meaning of a τ_{CBSD}-structure conforming to the axioms of Φ_{CBSD}. In contrast, latter UML diagrams, for example in Sec. 5.4, will be understood as *models of systems* — these diagrams will *not* be annotated.

Small boxes contain for every element of the diagram a unique identifier that represents the element in the structure. A simple package hierarchy is depicted that consists of the packages P, P1, and P2 that contain some components and interfaces. Table 5.2 lists relation symbols of τ_{CBSD} and how they are interpreted in the system S depicted in Fig. 5.9[4]. The containment as

[4]For reasons of simplicity, we omit here the relations for *name* and *qualifiedName*. Tuples of these relations

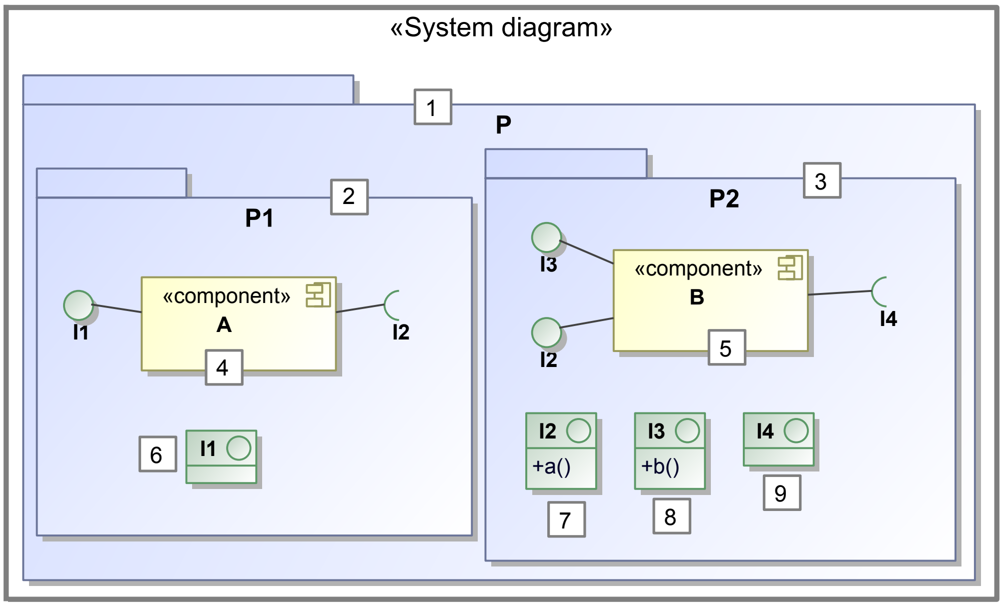


Figure 5.9: Exemplary system S according to τ_{CBSD}, Part 1.

Relation symbol r	Interpreting relation r^S in the system S
$Package$	$\{1,2,3\}$
$Component$	$\{4,5\}$
$Interface$	$\{6,7,8,9\}$
$containsPkg$	$\{(1,2),(1,3)\}$
$containsComponent$	$\{(2,4),(3,5)\}$
$containsInterface$	$\{(2,6),(3,7),(3,8),(3,9)\}$
$providesInterface$	$\{(4,6),(5,7),(5,8)\}$
$requiresInterface$	$\{(4,8),(5,9)\}$

Table 5.2: Relations of the example from Fig. 5.9.

well as the provide and require relationships are reflected by the depicted binary relations.

5.3.3. Structure Specification of Object-Oriented Classifiers (τ_{oos})

Object-oriented classifiers are interfaces and classes. Both have in common that they define members and signatures for methods, and both can inherit properties from interfaces or classes by extending them. These commonalities are reflected in a common relation symbol *OOClassifier*.

would for example be $(2, \text{“}P1'') \in name$ and $(2, \text{“}P :: P1'') \in qualifiedName$, assuming P is the root package.

Interfaces/Object-Oriented Classifiers

The signatures defined by a classifier are modelled by *definesSignature* $\subseteq$ *OOClassifier* $\times$ *Signature*, as depicted in Fig. 5.8. A signature is basically the name of an operation and a list of typed parameters. The name is reflected by the relation symbol *name* $\subseteq$ *NamedElement* $\times$ *String* [5]. The list of parameters (*Parameter*) of a signature is formalized by the relation symbol *hasParameter* $\subseteq$ *Signature* $\times$ *Parameter* and *nextParameter* $\subseteq$ *Parameter* $\times$ *Parameter* that defines a strict total order on the parameters for a fixed signature.

There are several axioms in Φ_{CBSD} that ensure correct relations between parameters and signatures. First, parameters related by *nextParameter* must belong to the same signature:

$$\begin{aligned}\mathbf{CBSD_1} &:= \forall p \forall q : \mathit{nextParameter}(p, q) \rightarrow \\ &\exists s : (\mathit{hasParameter}(s, p) \wedge \mathit{hasParameter}(s, q))\end{aligned} \tag{5.10}$$

If a signature has parameters, there is exactly one without predecessor and exactly one without succeeding parameter:

$$\begin{aligned}\mathbf{CBSD_2} &:= \forall s : ((\neg\exists p : \mathit{hasParameter}(s, p)) \vee \\ &\exists! p : (\mathit{hasParameter}(s, p) \wedge (\neg\exists q : \mathit{nextParameter}(q, p))))\end{aligned} \tag{5.11}$$

$$\begin{aligned}\mathbf{CBSD_3} &:= \forall s : ((\neg\exists p : \mathit{hasParameter}(s, p)) \vee \\ &\exists! p : (\mathit{hasParameter}(s, p) \wedge (\neg\exists q : \mathit{nextParameter}(p, q))))\end{aligned} \tag{5.12}$$

The members defined by an interface (or a class) are modelled by the relation symbol *definesMember* $\subseteq$ *OOClassifier* $\times$ *Member*. Members are typed elements that can either refer to a primitive data value or an instance of a complex type if typed with an interface or a class.

Furthermore, interfaces can subtype each other, reflected by *extends* $\subseteq$ *OOClassifier* $\times$ *OOClassifier*, whereas $(i_1, i_2) \in \mathit{extends}$ indicates that i_1 is subtype of i_2. We furthermore define $\mathit{extends}^+$ as the transitive closure of *extends*. It hence contains all the tuples from *extends* as well as all further tuples that reflect indirect subtype relationships between object-oriented classifiers.

The extend relation can only exists between interfaces and interfaces, or between classes and classes. The relation symbol *implements* discussed later describes the relationship between an interface and the classes implementing it. Hence the following axiom is required:

$$\begin{aligned}\mathbf{CBSD_4} &:= \forall x \forall y : \mathit{extends}(x, y) \rightarrow (\mathit{Class}(x) \wedge \mathit{Class}(y)) \vee \\ &(\mathit{Interface}(x) \wedge \mathit{Interface}(y))\end{aligned} \tag{5.13}$$

The relation symbols *hasMember* $\subseteq$ *OOClassifier* $\times$ *Member* and *inheritsMember* $\subseteq$ *OOClassifier* $\times$ *Member* reflect inheritance at the internal, structural level of object-oriented classifiers; as described, *definesMember* models members that a classifier defines itself;

[5] *Signature*, as well as every other unary relation symbol from Fig.5.8 except *MethodBody*, "subsets" *NamedElement*. See Appendix

inheritsMember describes which members are inherited; *hasMember* describes the union of both.

The following constraints have thus to hold:

$$\begin{aligned}\mathbf{CBSD_5} := \forall x \forall y : ((\mathit{definesMember}(x,y) \rightarrow \neg\, \mathit{inheritsMember}(x,y)) \wedge \\ (\mathit{inheritsMember}(x,y) \rightarrow \neg\, \mathit{definesMember}(x,y)))\end{aligned} \tag{5.14}$$

$$\begin{aligned}\mathbf{CBSD_6} := \forall x \forall y : (\mathit{hasMember}(x,y) \leftrightarrow \\ (\mathit{inheritsMember}(x,y) \vee \mathit{definesMember}(x,y)))\end{aligned} \tag{5.15}$$

$$\begin{aligned}\mathbf{CBSD_7} := \forall x \forall y : (\mathit{inheritsMember}(x,y) \rightarrow \\ \exists z : ((\mathit{extends}(x,z) \vee \mathit{implements}(x,z)) \wedge \mathit{hasMember}(z,y)))\end{aligned} \tag{5.16}$$

Analogously, constraints are defined for inheritance regarding method signatures:

$$\begin{aligned}\mathbf{CBSD_8} := \forall x \forall y : ((\mathit{definesSignature}(x,y) \rightarrow \neg\, \mathit{inheritsSignature}(x,y)) \wedge \\ (\mathit{inheritsSignature}(x,y) \rightarrow \neg\, \mathit{definesSignature}(x,y)))\end{aligned} \tag{5.17}$$

$$\begin{aligned}\mathbf{CBSD_9} := \forall x \forall y : (\mathit{hasSignature}(x,y) \leftrightarrow \\ (\mathit{inheritsSignature}(x,y) \vee \mathit{definesSignature}(x,y)))\end{aligned} \tag{5.18}$$

$$\begin{aligned}\mathbf{CBSD_{10}} := \forall x \forall y : (\mathit{inheritsSignature}(x,y) \rightarrow \\ \exists z : ((\mathit{extends}(x,z) \vee \mathit{implements}(x,z)) \wedge \mathit{hasSignature}(z,y)))\end{aligned} \tag{5.19}$$

Running Example: Adding Interface Details

In Fig. 5.10, the interfaces IF2 and IF3 from Fig. 5.9 are further refined. We can see, that IF3 subtypes IF2 and that IF3 defines an additional signature. Table 5.3 lists the relation symbols and the assigned relations for the cutout of S depicted in 5.10.

Notice that return parameters are named with “return” which cannot be explicitly seen in the graphical notation. Furthermore, it can be seen that $\mathit{nextParameter}^S$ defines the order of the parameters of a correctly regarding the axioms mentioned above.

Classes

Classes are types that are local to components. They cannot be accessed directly from outside the component. Their semantics is apart from that identical to that one common in object-orientation. Classes provide methods by providing method bodies (*MethodBody*) for signatures

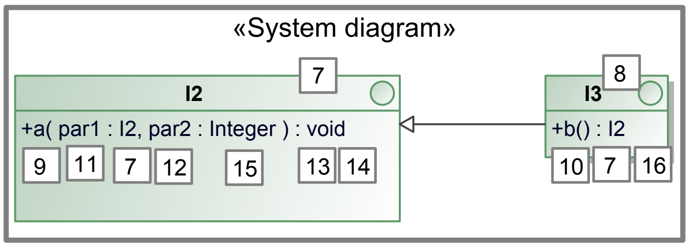


Figure 5.10: Example of Fig. 5.9 further refined.

Relation symbol r	Interpreting relation r^S in the system S
Interface	$\{\ldots, 7, 8, \ldots\}$
Signature	$\{9, 10\}$
Parameter	$\{11, 12, 13, 16\}$
PrimitiveType	$\{14, 15\}$
extends	$\{(8, 7)\}$
definesSignature	$\{(7, 9), (8, 10)\}$
inheritsSignature	{(8,9)}
hasSignature	{(7,9),(8,9),(8,10)}
hasParameter	$\{(9, 11), (9, 12), (9, 13), (10, 16)\}$
name	$\{\ldots, (13, \text{“return”}), (16, \text{“return”}), \ldots\}$
nextParameter	$\{(11, 12), (12, 13)\}$
hasType	$\{\ldots, (11, 7), (12, 15), (13, 14), (16, 7), \ldots\}$

Table 5.3: Relations of the example from Fig. 5.10.

defined in the class itself, in one of its super classes, or in an implemented interface. The provided methods can also either be defined in the class itself or be inherited from extended classes. This is expressed by the relation symbols *hasMethod*, *definesMethod*, and *inheritsMethod* (see Fig. 5.8) and the following axioms:

$$\begin{aligned}\mathbf{CBSD_{11}} := \forall x \forall y : ((\mathit{definesMethod}(x, y) \rightarrow \neg\, \mathit{inheritsMethod}(x, y)) \wedge \\ (\mathit{inheritsMethod}(x, y) \rightarrow \neg\, \mathit{definesMethod}(x, y)))\end{aligned} \tag{5.20}$$

$$\begin{aligned}\mathbf{CBSD_{12}} := \forall x \forall y : (\mathit{hasMethod}(x, y) \leftrightarrow \\ (\mathit{inheritsMethod}(x, y) \vee \mathit{definesMethod}(x, y)))\end{aligned} \tag{5.21}$$

$$\begin{aligned}\mathbf{CBSD_{13}} := \forall x \forall y : (\mathit{inheritsMethod}(x, y) \rightarrow \\ \exists z : ((\mathit{extends}(x, z) \vee \mathit{implements}(x, z)) \wedge \mathit{hasMethod}(z, y)))\end{aligned} \tag{5.22}$$

The signature of a method body is assigned by $\mathit{implementsSignature} \subseteq \mathit{MethodBody} \times \mathit{Signature}$. The signature of method has of course to be part of the class which defines the method:

$$\begin{aligned}\mathbf{CBSD_{14}} &:= \forall x \forall y : (\mathit{definesMethod}(x, y) \leftrightarrow \\ &\quad \exists s : (\mathit{hasSignature}(x, s) \wedge \mathit{implementsSignature}(y, s)))\end{aligned} \tag{5.23}$$

Not only classes are local to components, also single inheritance hierarchies are completely contained in single components. A class inheriting another class defined in a different component would violate the basic principle that components can only be connected and depend on each other by the usage of interfaces. This means that the following constraint has to hold in every component-based system:

$$\begin{aligned}\mathbf{CBSD_{15}} &:= \forall x \forall y : (\mathit{Class}(x) \wedge \mathit{extends}(x, y) \rightarrow \\ &\quad \exists z : (\mathit{containsClass}(z, x) \wedge \mathit{containsClass}(z, y)))\end{aligned} \tag{5.24}$$

A class can implement interfaces ($\mathit{implements} \subseteq \mathit{Class} \times \mathit{Interface}$), as mentioned above. In this case, the class must provide a method for every signature defined in the interface or one of its supertypes. The relation $\mathit{implements}^{+} \subseteq \mathit{Class} \times \mathit{Interface}$ contains all relations between interfaces and implementing classes and includes also indirect implementations that arise from classes implementing subtypes of interfaces. Hence, if a class c implements an interface i_2 which is a subtype of i_1, c also indirectly implements i_1. More formally it has to hold that $(c, i_2) \in \mathit{implements}$ and $(i_2, i_1) \in \mathit{extends}$ implies that $(c, i_1) \in \mathit{implements}^{+}$. This is formally captured in Φ_{CBSD} by the axiom

$$\begin{aligned}\mathbf{CBSD_{16}} &:= \forall c \forall i : (\mathit{implements}^{+}(c, i) \leftrightarrow \\ &\quad \mathit{implements}(c, i) \vee (\exists j : \mathit{extends}^{+}(j, i) \wedge \mathit{implements}(c, j)) \vee \\ &\quad (\exists d : \mathit{extends}^{+}(c, d) \wedge (\mathit{implements}(d, i) \vee \\ &\quad\ (\exists j : \mathit{extends}^{+}(j, i) \wedge \mathit{implements}(d, j)))))\end{aligned} \tag{5.25}$$

Running Example: Adding Classes

Figure 5.11 refines component B from Fig. 5.9. The component contains three classes C1, C2, and C3 whereas C2 subclasses C1 and defines another method d. Moreover, C1 defines a member that is typed with C3.

Table 5.4 shows the relations modelling the example of Fig. 5.11. The containment relation between B and its classes is modelled by $\mathit{containsClass}^{S} = \{(5, 23), (5, 24), (5, 25)\}$. $\mathit{implements}^{S}$ shows that C1 and C2, which are the entities with $\mathit{id} = 23$ and $\mathit{id} = 24$, respectively, implement I3 ($\mathit{id} = 8$) directly and I2 ($\mathit{id} = 7$) indirectly. Both classes provide method bodies for the signatures defined in the interfaces, as defined by $definesMethod^{S}$. Because of $(24, 19), (24, 20) \in definesMethod^{S}$, it is also specified that C2 overwrites the implementation of C1 for a(...) and b(), since it obviously do not inherit the method bodies of $C1$.

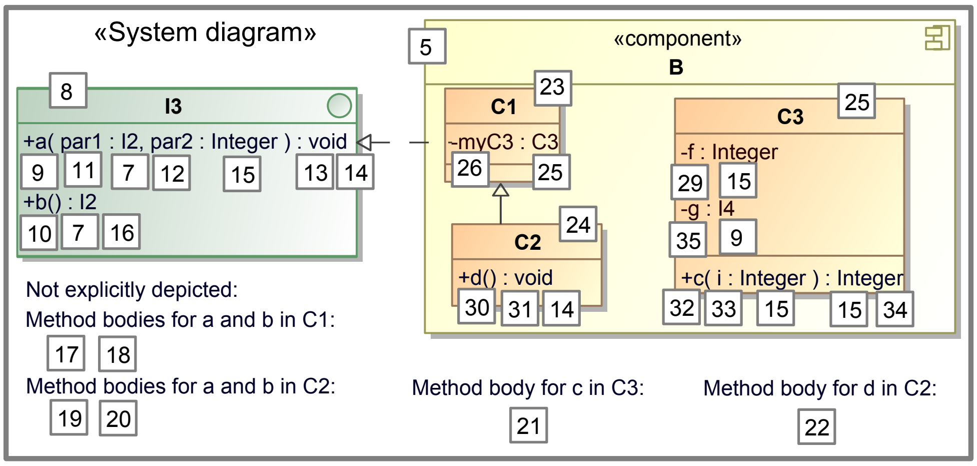


Figure 5.11: The example of Fig. 5.10 extended by classes.

Relation symbol r	Interpreting relation r^S in the system S
$Signature$	$\{9, 10, 30, 32\}$
$Parameter$	$\{11, 12, 13, 16, 31, 33, 34\}$
$PrimitiveType$	$\{14, 15\}$
$Class$	$\{23, 24, 25\}$
$MethodBody$	$\{17, 18, 19, 20, 21, 22\}$
$Member$	$\{26, 29, 35\}$
$containsClass$	$\{(5, 23), (5, 24), (5, 25)\}$
$definesSignature$	$\{(7, 9), (8, 10), (24, 30), (25, 32)\}$
$inheritsSignature$	$\{(8, 9), (23, 9), (23, 10), (24, 9), (24, 10)\}$
$hasSignature$	$definesSignature^S \cup inheritsSignature^S$
$definesMember$	$\{(23, 26), (25, 29), (25, 35)\}$
$inheritsMember$	$\{(24, 26)\}$
$hasMember$	$definesMember^S \cup inheritsMember^S$
$definesMethod$	$\{(23, 17), (23, 18), (24, 19), (24, 20), (24, 22), (25, 21)\}$
$implements$	$\{(23, 8)\}$
$implements^+$	$\{(23, 8), (24, 8), (23, 7), (24, 7)\}$
$extends$	$\{(8, 7), (24, 23)\}$
$extends^+$	$\{(8, 7), (24, 23)\}$
$hasType$	$\{(26, 25), (29, 15), (35, 9)\}$

Table 5.4: Relations of the example from Fig. 5.11.

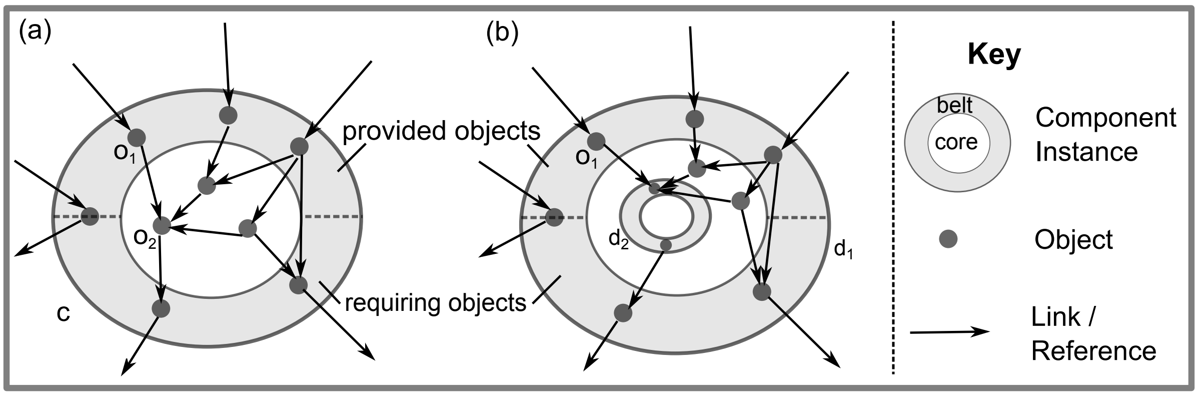


Figure 5.12: Atomic (a) and hierarchical (b) component instances.

5.3.4. Structure Specification of Components and Systems (τ_{cs})

It is useful to understand the structures of component-based systems at runtime in order to introduce the concepts that are required to specify the internal structure of components and the systems they form. The structural elements of object-oriented systems build the foundations for the structure of component-based systems. These are mainly objects and references, also known as links. Objects are instances of classes. Objects can send messages to each other if there are references, or links, between them. The message receiving object reacts by invoking a corresponding method, either synchronously or asynchronously.

Objects are hence relatively simple constructs. Their states consist of the values of their attributes and the set of objects they are linked to [Bal99]. Components as an extension of object-orientation allow defining complex instances that are build of arbitrarily complex networks of objects. Instantiating a component includes creating the specified network of objects, as well as destroying the instance means breaking the network down. Furthermore, the definition of provided and required interfaces for components enables the designer to specify, how the instances of a component may be accessed by other components or the system's environment, and which parts it needs to access itself.

Part (a) of Fig. 5.12 shows a simple component instance. The core of the component instance *c* contains objects that are not accessible from any other object outside of *c*. All links to those objects are instead pointing from or to objects in the outer belt of *c*, or remain in the inner core. The upper part of the belt contains the only objects, the provided objects, which can be accessed by the environment of *c*. Those object themselves may access objects in the inner core of *c*. The lower half of the belt contains the only objects of *c*, the requiring objects that are allowed to have links to objects in the environment of *c*. Of course, an object of the belt can be both, provided and requiring, at the same time.

It is important to understand that, as contribution to the concept of encapsulation, a client of the component does not know the class of a provided object it would like to access. Instead, it only knows the interface implemented by the class and which must be one of the provided interfaces of the component. This applies similarly to requiring objects. Their classes require an interface that is in the component's set of required interfaces. At runtime, this requirement relation is satisfied by a link between the requiring object and some instance of a class implementing that interface.

Furthermore, instances of components can be used as substructure of instances of other components. This way, hierarchical components can be created. Part (b) of Fig. 5.12 shows an example. The component instance d_1 is basically the same as c with the difference that the atomic object o_2 from the core of c is replaced by a complex component instance d_2. For d_2, we have to adhere to the same constraints as for c — accesses *to* the component are only allowed via provided objects, accesses *by* the component only via requiring objects. Hence, the object o_1, which has had a link to object o_2 in case (a), has now a link to a provided object of d_2.

Like a class defines the attributes that instances have or the methods that can be called at them, a component has to specify which provided and requiring objects, and which "inner objects" form an instance of the component. The specification elements used for this task are *ports*, *inner parts*, and *connectors*. They and their formalization as relation symbols of $\tau_{cs} \subset \tau_{CBSD}$, the signature subset used for the specification of the internal component structures, are introduced in the following subsection.

Ports, Inner Parts, and Connectors

The elements that are used to specify the inner structure of components are called *parts*. Parts are named and typed elements. The name can be understood as a role. At runtime, an instance of the type assigned to a part will play this role and can be addressed by this name. For example, let us assume that we have an information system with a data access component. Inside this component, we could model two parts named "primaryDB" and "backupDB" both of type "Database", which could be a complex component itself.

Two kinds of parts are used to specify which objects constitute a component instance at runtime. *Inner parts* define which objects are required in the core of a component. This means that the type of an inner part defines the type of an object that plays the role given by the name of the part. *Ports* do the same for objects that are either provided or requiring objects. Ports for provided objects are called *provided ports*. At runtime, for each provided port an instance of the given type exists that can be accessed from the environment of the component instance. On the other hand, *required ports* are ports specifying links to required objects from the environment of a component.

Figure 5.13 depicts the relation symbols of τ_{CBSD} that are used to formalize parts. All parts are collected by the relation symbol *Part*. Parts as discussed so far are furthermore refinements of the relation symbol *SinglePart* in contrast to *connectors* that will be discussed later. Single parts represent roles that can be played by instances of classes or components. *Port* and *InnerPart* refine *SinglePart* for the different kind of parts as mentioned above. Ports are further distinguished into *ProvidedPort* and *RequiredPort*. The relation $\mathit{encapsulates} \subseteq \mathit{Component} \times \mathit{Part}$ assigns parts to the component that contains them.

All parts can be typed, since $\mathit{Port} \subseteq \mathit{TypedElement}$. It is important to notice that in the case of provided ports the type assigned by the *hasType* relation is *not* the type that a client sees when linking to the object playing the role of the port. Instead, an interface is provided by the port which is the type visible to the environment. This is represented by the relation symbol $\mathit{providedInterface} \subseteq \mathit{ProvidedPort} \times \mathit{Interface}$. The following constraints have to hold with regard to provided ports:

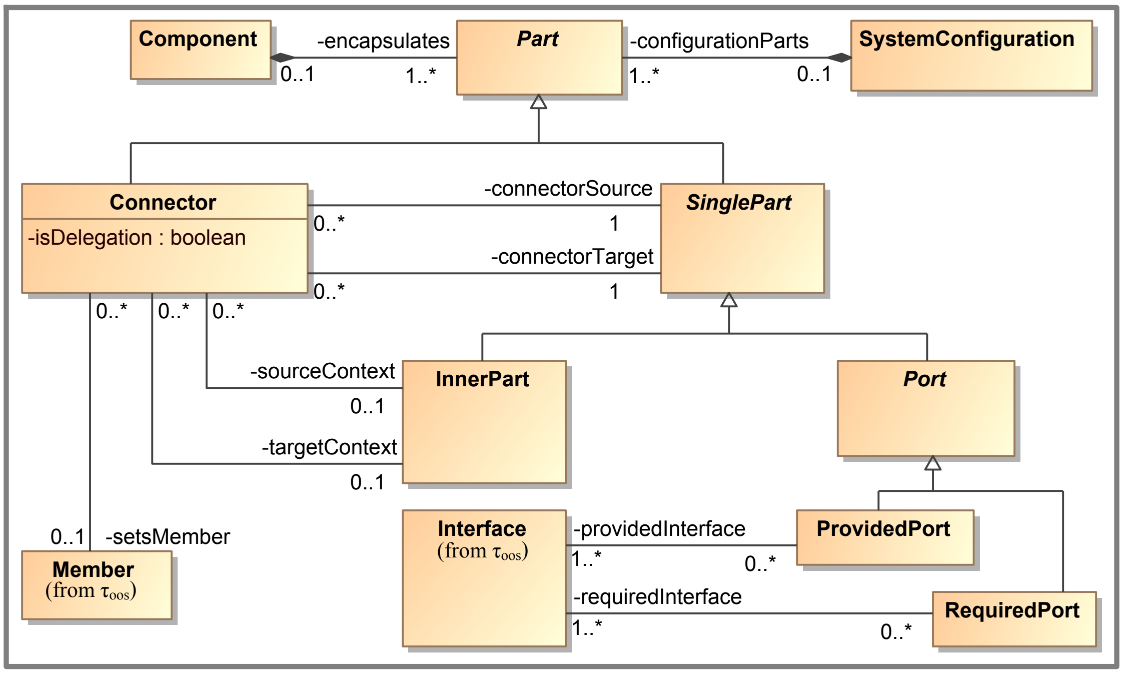


Figure 5.13: Relation symbols in τ_{cs} to model parts.

- Only a class defined by the encapsulating component can be type of a port:

$$\begin{aligned}\textbf{CBSD}_{17} := \forall x \forall y \forall z : (&Port(x) \wedge Component(y) \wedge Type(z) \wedge \\ &encapsulates(y, x) \wedge hasType(x, z) \rightarrow \\ &Class(z) \wedge containsClass(y, z))\end{aligned} \tag{5.26}$$

  This also holds for required ports.

- The provided interface of a provided port must be in the set of provided interfaces of the encapsulating component:

$$\begin{aligned}\textbf{CBSD}_{18} := \forall x \forall y \forall z : (&ProvidedPort(x) \wedge Component(y) \wedge Interface(z) \wedge \\ &encapsulates(y, x) \wedge providedInterface(x, z) \rightarrow \\ &providesInterface(y, z))\end{aligned} \tag{5.27}$$

- The class being the type of a provided port has to implement the provided interface of the port:

$$\begin{aligned}\textbf{CBSD}_{19} := \forall x \forall y \forall z : (&Class(x) \wedge ProvidedPort(y) \wedge Interface(z) \wedge \\ &hasType(y, x) \wedge providedInterface(y, z) \rightarrow \\ &implements^{+}(x, z))\end{aligned} \tag{5.28}$$

Required ports express that a component instance encapsulates objects that depend on objects outside of the scope of the component. By the relation symbol *requiredInterface* $\subseteq$

RequiredPort × *Interface* the type of such a required object is determined. Similar to provided ports, the required interface of a port has also to be required by the component:

$$\begin{aligned}\mathbf{CBSD_{20}} := \forall x \forall y \forall z : (&\mathit{RequiredPort}(x) \land \mathit{Component}(y) \land \mathit{Interface}(z) \land \\ &\mathit{encapsulates}(y, x) \land \mathit{requiredInterface}(x, z) \rightarrow \\ &\mathit{requiresInterface}(y, z))\end{aligned} \tag{5.29}$$

Inner parts represent roles played by instances of the component core. Those instances cannot be accessed directly by other component instances. Thus, inner parts do not need a distinction between their visible and their actual type.

In contrast to ports, inner parts can also be typed by components, not just classes, since instances in the core of a component can also be component instances. This difference is axiomatically specified by the constraint for single parts

$$\begin{aligned}\mathbf{CBSD_{21}} := \forall x \forall y : (&\mathit{SinglePart}(x) \land \mathit{Type}(y) \land \mathit{hasType}(x, y) \rightarrow \\ &\mathit{Class}(y) \lor \mathit{Component}(y))\end{aligned} \tag{5.30}$$

and the port specific axioms mentioned above.

Objects can be linked, and they can play roles defined by ports or inner parts, depending on whether they are provided, requiring, or “inner” objects of a component instance. Obviously, it should also be possible to specify that a link has to exist between two objects playing dedicated roles. This is done by the usage of *connectors*. Connectors connect a source part with a target part indicating, that the object playing the role of the source part has a reference playing the role of the target part. The use of connectors enables us to specify the “edges” of the network of instances that is created when instantiating a component.

Three kinds of use cases for connectors can be distinguished that correspond to the three kinds of generally allowed linkings between objects in components:

- *Connectors between inner parts* are used to specify links between objects of the component core.
- *Connectors between provided ports and inner parts* specify links between required objects and core objects.
- *Connectors between inner parts/provided ports and required ports* specify links between core objects/provided objects and requiring objects, or allow the *delegation* of links.
- *Connectors marked as delegators* connect ports with the interior of a component indicating that links from the environment *to* the component, or links *from* the component to the environment are delegated.

To understand the delegation of links, it is useful to understand how component instances can be connected by connectors, since connectors are only able to specify links between objects. Consider p, being typed by component A, being a inner part of component C. A has a required port x with an interface I that is provided by the port y of a component B. There is another part q of component A that is typed with B. In this scenario, there will be a requiring object at runtime, encapsulated by the component instance playing the role p, that play the role of x and requires an instance of I. On the other hand, the instance playing the role of B encapsulates such an object, that one playing the role of y. To specify, that those both ports should be linked, we can specify a connector between them, defining p and q as the *context* of it. The context of connector is required because ports are defined for types, namely components, and not for specific inner parts. A port is hence defined only once per type (component). This means that a connector between the two ports above would indicate, that every instance of A is linked with an instance of B, not only in the context of playing certain roles (p and q) in the inner structure of C.

Since component instances can play the roles of inner parts, it can happen that nested requiring objects are contained in a component core. Furthermore, it can happen that such a requiring object cannot be satisfied directly because it cannot be linked to an object of the surrounding component's core. The reason for this is, that the surrounding component does not necessarily need to implement the required interface but requires it itself. Thus, an object that can be linked to, must be found outside of the surrounding component. To specify such circumstances, delegate connectors are used that connect a required port of an inner part's type with a required port of the outer component. References of nested requiring objects can this way be delegated to the outside.

Connectors are formalized in τ_{CBSD} as members of the relation symbol *Connector* (see Fig. 5.13). Connectors are considered to be parts themselves since they indicate roles and are encapsulated by components. They connect single parts as introduced before which is modelled by the relation symbols $connectorSource \subseteq Connector \times SinglePart$ and $connectorTarget \subseteq Connector \times SinglePart$. The context of connectors is formalized by the relation symbols $sourceContext \subseteq Connector \times InnerPart$ and $targetContext \subseteq Connector \times InnerPart$.

Some general constraints hold for connectors. First of all, component-typed parts cannot be source or target of connectors but only class-typed ones:

$$\begin{aligned}\mathbf{CBSD_{22}} := \forall x \forall y \forall z : (&Connector(x) \wedge SinglePart(y) \wedge SinglePart(z) \wedge \\ &connectorSource(x,y) \wedge connectorTarget(x,z) \rightarrow \\ &\exists t \exists t' : Class(t) \wedge Class(t') \wedge hasType(y,t) \wedge hasType(z,t'))\end{aligned} \tag{5.31}$$

For all connectors holds that source and target context must be parts that are encapsulated in the same component as the connector:

$$\begin{aligned}\mathbf{CBSD_{23}} := \forall x \forall y \forall z \forall z' : (&Connector(x) \wedge Component(y) \wedge SinglePart(z) \wedge \\ &SinglePart(z') \wedge encapsulates(y,x) \wedge sourceContext(x,z) \wedge \\ &targetContext(x,z') \rightarrow encapsulates(y,z) \wedge encapsulates(y,z'))\end{aligned} \tag{5.32}$$

Each connector has to correspond to one of the three cases mentioned above — this means it either has as context two inner parts, or a provided port and an inner part, or an inner part and a

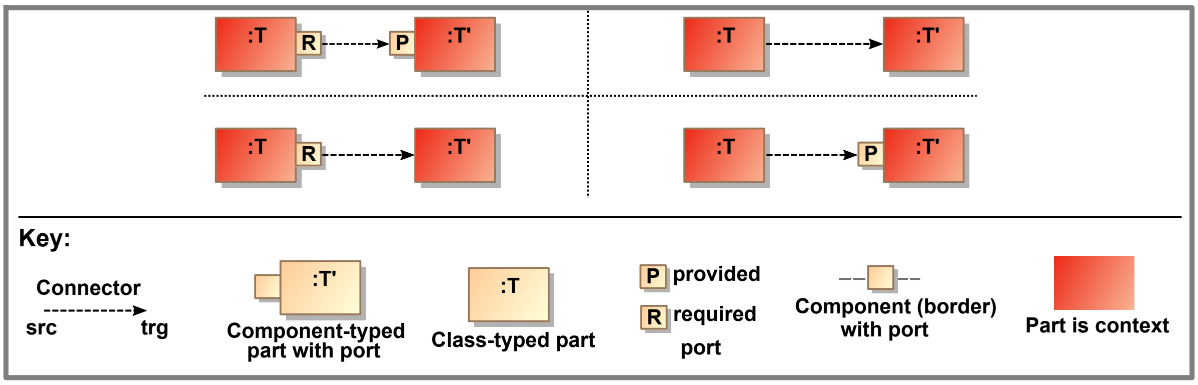


Figure 5.14: Possible connectors between inner parts.

required port. The following axiom enforces these rules and excludes all other combinations:

$$\begin{aligned}\mathbf{CBSD_{24}} := \forall x \forall y \forall z : (&Connector(x) \wedge sourceContext(x,y) \wedge targetContext(x,z) \rightarrow \\ &(InnerPart(y) \wedge InnerPart(z)) \vee \\ &(ProvidedPort(y) \wedge InnerPart(z)) \vee \\ &(InnerPart(y) \wedge RequiredPort(z)))\end{aligned} \tag{5.33}$$

If a connector has a context typed by a component, the corresponding end, either source or target, must refer to a port that is defined for the type of the context under consideration:

$$\begin{aligned}\mathbf{CBSD_{25}} := \forall x \forall y \forall z : (&Connector(x) \wedge SinglePart(y) \wedge Component(z) \wedge \\ &sourceContext(x,y) \wedge hasType(y,z) \rightarrow (\exists p : Port(p) \wedge \\ &connectorSource(x,p) \wedge encapsulates(z,p)))\end{aligned} \tag{5.34}$$

$$\begin{aligned}\mathbf{CBSD_{26}} := \forall x \forall y \forall z : (&Connector(x) \wedge SinglePart(y) \wedge Component(z) \wedge \\ &targetContext(x,y) \wedge hasType(y,z) \rightarrow (\exists p : Port(p) \wedge \\ &connectorTarget(x,p) \wedge encapsulates(z,p)))\end{aligned} \tag{5.35}$$

A non-delegating connector defines that a link will be established at runtime between the source and the target that the connector is pointing to. A delegating connector points to the required port that delegates the link. In each case, the corresponding link is defined by a member of the source part. The relation symbol $setsMember \subseteq Connector \times Member$ models which member is actually set by a connector. It defines that, by specifying the connector, the referenced member is set to the object that the connector points to[6]. At the same time, the type

[6]In case of a delegating connector, the target of the connector does not refer directly to a target object but to an outer required port which is used for delegation.

of this member is also the type of the connector.

$$\begin{aligned}\mathbf{CBSD_{27}} := \forall x \forall y \forall z : (&Connector(x) \land SinglePart(y) \land Member(z) \land \\ &connectorSource(x, y) \land setsMember(x, z) \rightarrow \\ &(\forall t : hasType(y, t) \rightarrow hasMember(t, z)) \land \\ &(\forall t : hasType(z, t) \leftrightarrow hasType(x, t)))\end{aligned} \tag{5.36}$$

The type of the target has to conform to the type of the connector:

$$\begin{aligned}\mathbf{CBSD_{28}} := \forall x \forall y \forall z : (&Connector(x) \land SinglePart(y) \land connectorTarget(x, y) \rightarrow \\ &(\forall t \forall t' : hasType(y, t) \land hasType(x, t') \rightarrow \\ &(implements^{+}(t, t') \lor extends^{+}(t, t'))))\end{aligned} \tag{5.37}$$

There are four possible combinations of connecting inner parts by connectors resulting from the fact that source and target context can each be typed either by a class or a component. The combinations are depicted in Fig. 5.14. Both, the source context and the target context are always inner parts. The source of a connector is equal to the source context if and only if the source context is class-typed. It is a provided port of the source context's type if and only if the source context is component-typed. The same rule applies to the target context:

$$\begin{aligned}\mathbf{CBSD_{29}} := \forall x \forall y \forall z : &Connector(x) \land InnerPart(y) \land InnerPart(z) \land \\ &sourceContext(x, y) \land targetContext(x, z) \rightarrow \\ &(\exists p : connectorSource(x, p) \land (y = p \lor RequiredPort(p)))\end{aligned} \tag{5.38}$$

$$\begin{aligned}\mathbf{CBSD_{30}} := \forall x \forall y \forall z : &Connector(x) \land InnerPart(y) \land InnerPart(z) \land \\ &sourceContext(x, y) \land targetContext(x, z) \rightarrow \\ &(\exists p : connectorTarget(x, p) \land (z = p \lor ProvidedPort(p)))\end{aligned} \tag{5.39}$$

Two combinations are possible to connect a provided port of a component with an inner part — the latter is either class-typed or component-typed, as depicted in Fig. 5.15. The source context is hence a provided port, while the target context is an inner part. These cases can be characterized by the axiom:

$$\begin{aligned}\mathbf{CBSD_{31}} := \forall x \forall y \forall z : (&Connector(x) \land ProvidedPort(y) \land InnerPart(z) \land \\ &sourceContext(x, y) \land targetContext(x, z) \rightarrow connectorSource(x, y) \land \\ &(\exists p : connectorTarget(x, p) \land (z = p \lor ProvidedPort(p))))\end{aligned} \tag{5.40}$$

Similar applies to the possibilities of connecting an inner part with a required port (see Fig. 5.16). Again, the inner part can be either typed by a class or by a component:

$$\begin{aligned}\mathbf{CBSD_{32}} := \forall x \forall y \forall z : (&Connector(x) \land InnerPart(y) \land \\ &RequiredPort(z) \land sourceContext(x, y) \land targetContext(x, z) \rightarrow \\ &connectorTarget(x, z) \land (\exists p : connectorSource(x, p) \land \\ &(y = p \lor RequiredPort(p))))\end{aligned} \tag{5.41}$$

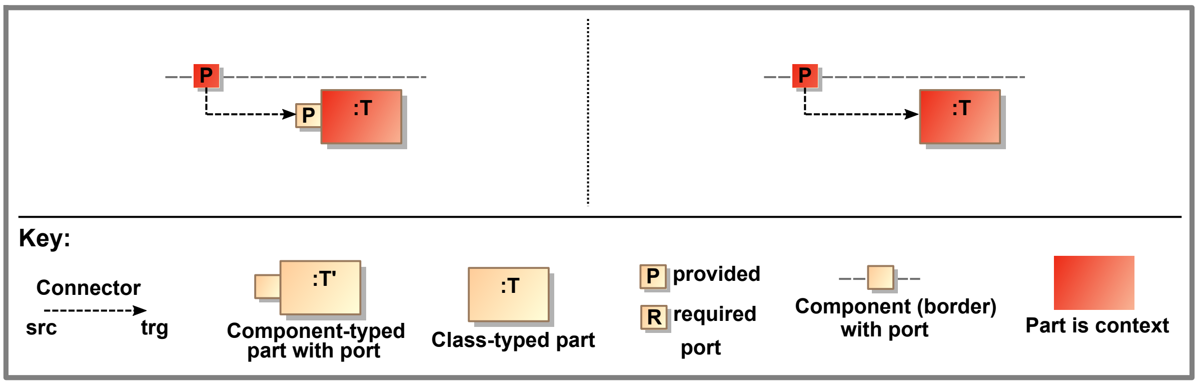


Figure 5.15: Possible connectors to connect provided ports with inner parts

Delegation connectors have to start at a provided port or to end at a required port:

$$\begin{aligned}\mathbf{CBSD_{33}} := \forall x : (&Connector(x) \wedge isDelegator(x, \text{“true”}) \rightarrow \\ &\exists y : (sourceContext(x, y) \rightarrow ProvidedPort(y)) \vee \\ &\exists y : (targetContext(x, y) \rightarrow RequiredPort(y)))\end{aligned} \tag{5.42}$$

Note that *isDelegator*(*x*, “true” distinguishes basically two kind of connectors common in many component models. Delegators support the hierarchical composition of components and systems by mapping externally visible elements to inner component parts (or vice versa). “Normal” connectors configure the links in the networks of objects representing a component instance.

In case a required port is target of a delegation connector, it must be rather understood as a reference variable pointing to an external object than an object pointing somewhere itself. Due to this different perspective, a required port cannot be addressed by normal and by delegation connectors at the same time:

$$\begin{aligned}\mathbf{CBSD_{34}} := \forall x : (&RequiredPort(x) \rightarrow ((\forall y : Connector(y) \wedge \\ &connectorTarget(y, x) \rightarrow isDelegator(x, \text{“true”})) \vee \\ &(\forall y : Connector(y) \wedge connectorTarget(y, x) \rightarrow isDelegator(x, \text{“false”}))))\end{aligned} \tag{5.43}$$

Required ports as targets of delegation connectors are therefore typed with the required interface, of which may only exists exactly one, of the port:

$$\begin{aligned}\mathbf{CBSD_{35}} := \forall x \forall y : &Connector(x) \wedge isDelegator(x, \text{“true”}) \wedge \\ &Interface(y) \wedge requiredInterface(x, y) \rightarrow \forall z : (hasType(x, z) \rightarrow z = y)\end{aligned} \tag{5.44}$$

Running Example: Adding Parts

Figure 5.17 continues the running example of this chapter and shows the inner structure of component B. There are two provided ports whereas ppo1 provides the interface I3 and ppo2

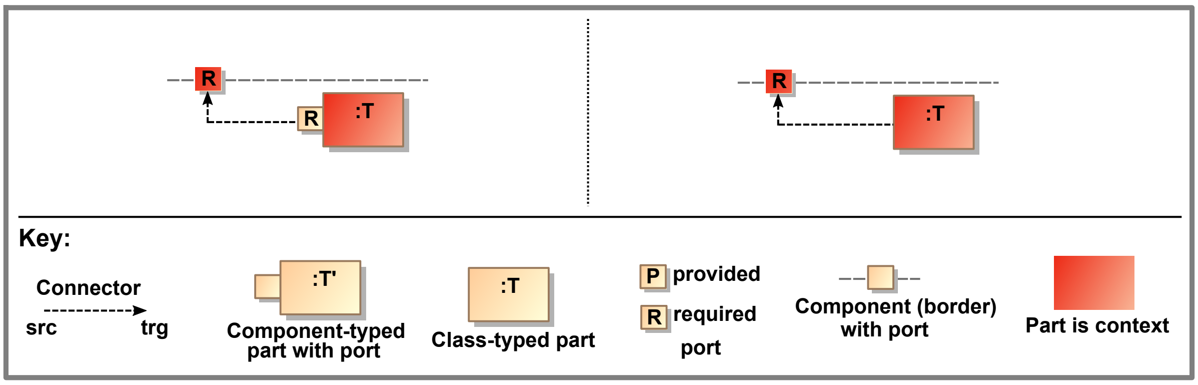


Figure 5.16: Possible connectors to connect inner parts with required ports

provides the interface I2. The type and realizing classifier of ppo1 is C1; the type of ppo2 is C2. Both ports are connected with the inner part pa1 which is typed by the class C3 (see Fig. 5.11). The connector con1 and con2 both set the corresponding member myC3, defined in C1 (see Fig. 5.11), i.e. the attribute refers to pa1 for both provided ports. As the definition of C3 shows, there is member g typed with I4 which has to refer to some object at runtime. Figure 5.17 shows that this reference is delegated to the required port rp1 of B.

Table 5.5 shows the relations for the addition made in Fig. 5.17 to the running example. New entities are parts, ports as well as inner parts and connectors. They are all encapsulated in component B as modelled by the relation $encapsulates^S$, and the types of all four single parts are set by $hasType^S$. The relationships between ports and their required and provided interfaces, respectively, are reflected by $providedInterface^S$ and $requiredInterface^S$. For example ppo1 providing I3 is reflected by $(38, 8) \in providedInterface^S$. Sources and targets of connectors are modelled by $connectorSource^S$ and $connectorTarget^S$. For example, the source of con1 is modelled by $(43, 38) \in connectorSource^S$; its target is modelled by $(43, 41) \in connectorTarget^S$.

Context information is correctly modelled for all connectors by $sourceContext^S$ and $targetContext^S$. Due to the fact that all source and target are either class-typed parts or ports "on the outer frame" of a component, those context relations contain the same elements as $connectorSource^S$ and $connectorTarget^S$, respectively.

System Configurations

A component-based system at runtime consists of instances of components, which encapsulate instances of classes. Component instances may enter and leave the system and the wiring between component instances can change. The initial state of the system, i.e. the instances of components and the links between them, is specified by *system configurations*. Similar to the inner structure of components, the structure of a system can be defined by parts specifying that an instance of a specific type plays a certain role in the system. In contrast to parts as elements of component specifications, parts used in system configurations cannot be typed by other types than components and connectors.

As can be seen in Fig. 5.13, system configurations are formalized by the rela-

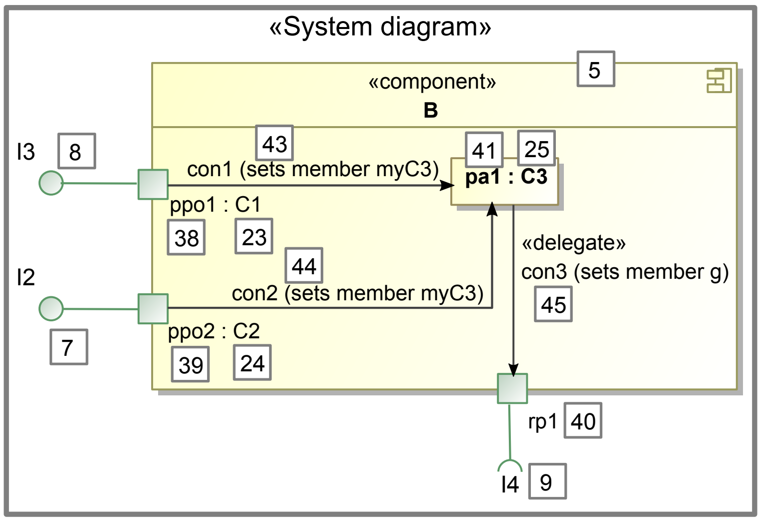


Figure 5.17: The inner structure of component *B* (Example of Fig. 5.11 refined).

Relation symbol r	Interpreting relation r^S in the system S
Component	$\{\dots,5,\dots\}$
Interface	$\{\dots,7,8,9,\dots\}$
Class	$\{23,24,25\}$
ProvidedPort	$\{38,39\}$
RequiredPort	$\{40\}$
InnerPart	$\{41\}$
Connector	$\{43,44,45\}$
encapsulates	$\{(5,38),(5,39),(5,40),(5,41),(5,42),(5,43),(5,44),(5,45)\}$
hasType	$\{\dots,(38,23),(39,24),(40,9),(41,25),\dots\}$
setsMember	$\{(43,26),(44,26),(45,25)\}$
providedInterface	$\{(38,8),(39,7)\}$
requiredInterface	$\{(40,9)\}$
connectorSource	$\{(43,38),(44,39),(45,41)\}$
connectorTarget	$\{(43,41),(44,41),(45,40)\}$
sourceContext	$\{(43,38),(44,39),(45,41)\}$
targetContext	$\{(43,41),(44,41),(45,40)\}$

Table 5.5: Relations of the example from Fig. 5.17.

tion symbol *SystemConfiguration*. The binary relation symbol *configurationPart* $\subseteq$ *SystemConfiguration* $\times$ *Part* contains tuples representing parts being elements of system configurations. The parts of a system configuration are either inner parts (typed by components) or connectors. Thus, it has to hold:

$$\begin{aligned}\mathbf{CBSD_{36}} :=\ & \forall x \forall y : (Part(x) \land SystemConfiguration(y) \land \\ & configurationPart(y,x) \rightarrow (InnerPart(x) \lor Connector(x)))\end{aligned} \tag{5.45}$$

The inner parts, as already mentioned, have to be component-typed:

$$\begin{aligned}\mathbf{CBSD_{37}} :=\ & \forall x \forall y : (InnerPart(x) \land SystemConfiguration(y) \land \\ & configurationPart(y,x) \rightarrow \forall z : (hasType(x,z) \rightarrow Component(z)))\end{aligned} \tag{5.46}$$

Of course, each part belongs either to a system configuration or to a component but not to both at the same time. This rule is enforced by the axiom:

$$\begin{aligned}\mathbf{CBSD_{38}} :=\ & \forall x : (Part(x) \rightarrow ((\exists y : encapsulates(y,x)) \leftrightarrow \\ & (\neg \exists y : configurationPart(y,x))))\end{aligned} \tag{5.47}$$

All axioms for single parts and connectors introduced in the previous section are also correct for parts as elements of system configuration. The only additional constraint is that parts being context of a connector in a system configuration, have to be in the same system configuration.

$$\begin{aligned}\mathbf{CBSD_{39}} :=\ & \forall x \forall y \forall z \forall z' : (SystemConfiguration(x) \land Connector(y) \land \\ & Part(z) \land Part(z') \land sourceContext(y,z) \land targetContext(y,z') \rightarrow \\ & (configurationPart(x,y) \leftrightarrow (configurationPart(x,z) \land \\ & configurationPart(x,z'))))\end{aligned} \tag{5.48}$$

With the relation symbols and the axioms introduced in this section, we are able to specify the static structure of component-based systems. In the following, we will introduce the symbols and axioms required to cover behavioural aspects.

5.3.5. Behaviour Specification (τ_{bhv})

The behaviour of a component-based system can be understood as the system's state change over time [Rau04]. The system state at any point during runtime consists of the state of its single component instances, which again is defined for each single component instance by the state of its internal objects. Their states are defined by the values of their attributes and their links to other objects. These states are changed by processing messages that are sent between objects; methods are executed and evaluate attribute values, modify links, create and destroy instances, and create and send new messages.

The behaviour of the system has to be specified at design-time. In our model of component-based systems, behaviour is specified by method bodies that define how an object of a given type reacts to the receiving of a certain message.

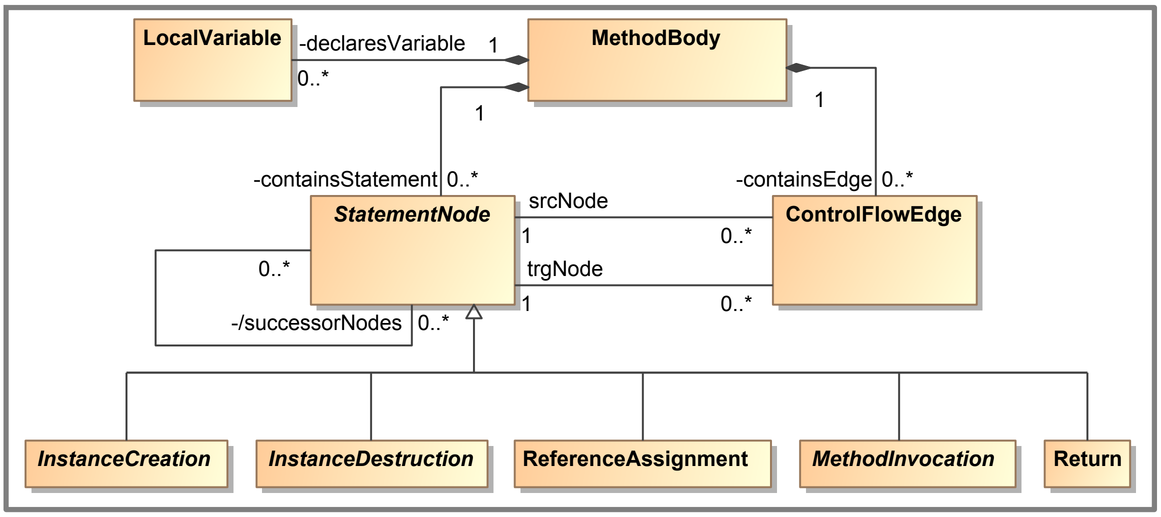


Figure 5.18: Formalizing control flow graphs.

Furthermore, it is assumed that the behaviour specification has an *imperative* character. This means that the specification consists of instruction statements that define how to change the state of the system; for example, a specification could describe that the value of an attribute is evaluated by sending a message to some object and passing the return value to the attribute.

Method bodies define hence a control flow describing the order of instruction statements by sequences, conditional branches, and loops. Their specific characteristics and syntax depend on the specification or programming language that is used.

Control flow graphs abstract from the specific language [All70]. Nodes represent statements and directed edges indicate a sequence between two nodes. Conditional branches manifest as nodes that have more than one outgoing edge; cycles in a graph indicate a loop.

Hence, it seems consequent to represent control flow graphs in τ_{CBSD} to reflect imperative behaviour specification formally. The following subsection introduces the representation of control flow graphs. After that, the subsequent subsection will describe different kinds of statements to create and destroy instances, to assign references, and to specify communication between objects.

Control Flow Graphs

A control flow graph is a directed graph with a dedicated root node from which each node is reachable. Nodes represent instruction statements, edges determine the order in which statements can be executed. Figure 5.18 depicts the relation symbols required to represent control flow graphs.

The relation symbols *StatementNode* and *ControlFlowEdge* model the nodes and edges of control flow graphs. They are contained in method bodies by the relation symbols $containsStatement \subseteq MethodBody \times StatementNode$ and $containsEdge \subseteq MethodBody \times ControlFlowEdge$. Edges connect source and target statement nodes indicating a possible sequence of statements in which the source statement is executed before the target statement. The corresponding relation symbols are $srcNode \subseteq ControlFlowEdge \times StatementNode$ and $trgNode \subseteq ControlFlowEdge \times StatementNode$.

Notice, that nodes are only instruction statements — the declaration of local variables is represented separately by the binary relation symbol $declaresVariable \subseteq MethodBody \times LocalVariable$.

The root nodes of control flow graphs are captured by the symbol $RootNode \subseteq StatementNode$. Each method body must contain exactly one root node:

$$\mathbf{CBSD_{40}} := \forall x : (MethodBody(x) \rightarrow \exists! y : RootNode(y) \wedge containsStatement(x, y)) \tag{5.49}$$

Control flow edges, of course, only connect nodes in the same method body:

$$\begin{aligned}\mathbf{CBSD_{41}} := \forall x, y, z, z' : (&ControlFlowEdge(x) \wedge MethodBody(y) \wedge containsEdge(y, x) \wedge \\ &StatementNode(z) \wedge StatementNode(z') \wedge srcNode(x, z) \wedge \\ &trgNode(x, z') \rightarrow containsStatement(y, z) \wedge containsStatement(y, z'))\end{aligned} \tag{5.50}$$

$successorNode \subseteq StatementNode \times StatementNode$ represents are all nodes that are directly connected to a node by outgoing control flow edges. It has to hold hence:

$$\begin{aligned}\mathbf{CBSD_{42}} := \forall x, y : (&successorNode(x, y) \leftrightarrow \exists z : ControlFlowEdge(z) \wedge \\ &srcNode(z, x) \wedge trgNode(z, y))\end{aligned} \tag{5.51}$$

The transitive closure $successorNode^+$ thus contains all direct or indirect successor nodes.

There exist different groups of instruction statements for different kinds of behaviour. These are statements for instance creation, instance destruction, reference assignment, method invocation and returning from methods. The corresponding relation symbols are subsetting *StatementNode* as depicted in Fig. 5.18.

Instance Creation

The structure specifications of components and system configurations (see Sec. 5.3.4) define the structures that are constructed when the corresponding component is instantiated, or the system is initialized. For each part, an instance is created. However, it is also possible to create instances during the execution of methods. The relation symbols for creation statements in the control flow graph are depicted in Fig. 5.19.

We distinguish between the creation of objects and the creation of component instances. Since objects can be seen as atomic instances, their creation is simpler than that of component instances. The latter implicitly includes the creation of instances for the parts that are defined for the component. Formally, there are two different relation symbols — *ObjectCreation* and *ComponentInstanceCreation* that both subset *InstanceCreation*. Both have in common that they refer to the classifier that is instantiated; this is reflected by the relation symbol

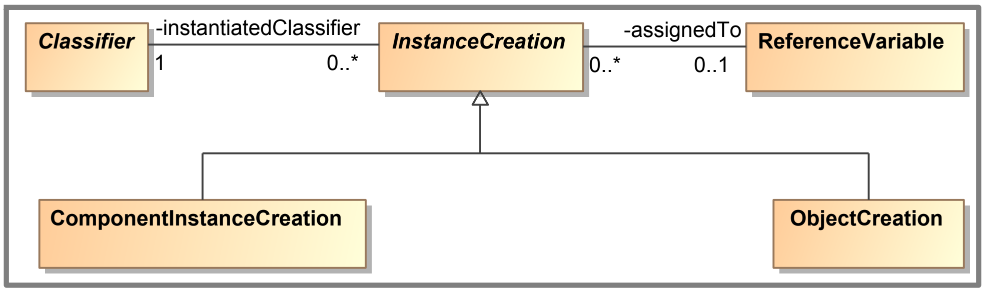


Figure 5.19: Formalization of instance creation statements.

$instantiatedClassifier \subseteq InstanceCreation \times Classifier$. Furthermore, both kinds of creation statement assign the newly created instance optionally to a reference variable — for example, to a local variable or to a member.

The instantiated classifier has of course to fit to the statement; an object creation statement cannot refer to a classifier that is a component. The constraints are the following axioms:

$$\begin{aligned}\mathbf{CBSD_{43}} := \forall x \forall y : (&ObjectCreation(x) \wedge Classifier(y) \wedge \\ &instantiatedClassifier(x,y) \rightarrow Class(y))\end{aligned} \tag{5.52}$$

$$\begin{aligned}\mathbf{CBSD_{44}} := \forall x \forall y : (&ComponentInstanceCreation(x) \wedge Classifier(y) \wedge \\ &instantiatedClassifier(x,y) \rightarrow Component(y))\end{aligned} \tag{5.53}$$

The reference variable to which the new instance is passed has to be type compatible. This means that the instantiated classifier, as far as it is a class, has to be a subtype of the reference variable type, or to implement it:

$$\begin{aligned}\mathbf{CBSD_{45}} := \forall x \forall y \forall z \forall z' : (&ObjectCreation(x) \wedge Classifier(y) \wedge \\ &instantiatedClassifier(x,y) \wedge assignedTo(x,z) \wedge hasType(z,z') \rightarrow \\ &implements^{+}(y,z') \vee extends^{+}(y,z'))\end{aligned} \tag{5.54}$$

Consider the following example, extending the classes depicted in Fig. 5.11. Let us assume, the method body of b in $C1$ declares a local variable v of type $C3$. Later, an instance of $C3$ is created and assigned to v. In pseudo code, this could look like:

```
I2 b() {
  C3 v; //declaration of v (id=50)
  v = new C3; //instance creation
  ...
}
```

This code snippet extends the relations of the running example as depicted in Tab. 5.6. The root node ($id = 51$), which is not explicitly specified in the pseudo-code above, is followed by

Relation symbol r	Interpreting relation r^S in the system S
MethodBody	$\{\ldots, 18, \ldots\}$ (see Fig. 5.11)
LocalVariable	$\{50\}$
StatementNode	$\{51, 52\}$
RootNode	$\{51\}$
ObjectCreation	$\{52\}$
ControlFlowEdge	$\{53\}$
Class	$\{25\}$ (see Fig. 5.11)
declaresVariable	$\{(18, 50)\}$
containsStatement	$\{(18, 51), (18, 52)\}$
containsEdge	$\{(18, 53)\}$
srcNode	$\{(53, 51)\}$
trgNode	$\{(53, 52)\}$
successorNode	$\{(51, 52)\}$
instantiatedClassifier	$\{(52, 25)\}$
assignedTo	$\{(52, 50)\}$
hasType	$\{\ldots, (50, 25), \ldots\}$

Table 5.6: Relations of the example from Fig. 5.11 extended by the described object creation.

the object creation statement ($id = 52$). The instance is assigned to the local variable v ($id = 50$), as $(52, 50) \in assignedTo^S$ indicates. The assignment is correct regarding type-compatibility, since the type of the v, denoted by $(50, 25) \in hasType^S$, is the same as the instantiated classifier, represented by $(52, 25) \in instantiatedClassifier^S$.

A component-based system will normally not just create instances but also establish links between instances (see Sec. 5.3.4). So far, we are only able to let a reference point to a newly created instance but not to change a reference towards an existing object.

Reference Assignment

The reference assignment, manifested by the relation symbol *ReferenceAssignment* is another kind of statement node. It reflects statements like

```
v=w
```

whereas v and w are type-compatible reference variables, and v is assigned the value of w; this means, after execution v is referring to the same instance as w. Figure 5.20 depicts how these statements are represented in τ_{CBSD}.

ReferenceAssignment is related to *ReferenceVariable* by two binary relation symbols. *passesValueFrom* refers to the reference variable on the right hand side of the assignment (w in the example); *passesValueTo* refers to the left hand side of the assignment, the reference whose value is changed.

Many programming languages define a null reference, a reference pointing to nothing. Uninitialized references normally point to “null”. Null can be assigned to references to indicate

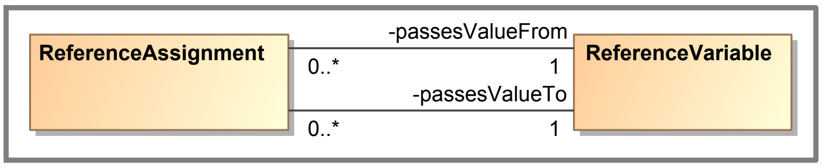


Figure 5.20: Reference assignment statements.

Relation symbol r	Interpreting relation r^S in the system S
LocalVariable	$\{50, 54\}$
StatementNode	$\{\dots, 55, \dots\}$
ReferenceAssignment	$\{55\}$
declaresVariable	$\{(18, 54)\}$
hasType	$\{\dots, (54, 25), \dots\}$
passesValueFrom	$\{(55, 54)\}$
passesValueTo	$\{(55, 50)\}$

Table 5.7: Relations of the example from Tab. 5.6 extended by the object creation described above.

that the reference does not refer to an instance any more. While null itself is not typed, it can be assigned to any reference variable. Null is defined as constant a symbol in τ_{CBSD} with the following constraints:

$$\mathbf{CBSD_{46}} := \neg\exists x : hasType(null, x) \tag{5.55}$$

$$\mathbf{CBSD_{47}} := \neg\exists x : (ReferenceAssignment(x) \land passesValueTo(x, null)) \tag{5.56}$$

Table 5.7 summarises the most important changes to the relations from Tab. 5.6 if the reference assignment above follows somewhere later in the body of the method b.

The local variable w ($id = 54$) is additionally declared. The statement node with $id = 55$ is the reference assignment that passes the value from w to v, as can easily be seen by looking at $passesValueFrom^S$ and $passesValueTo^S$.

Instance Destruction

The opposite statements of statements for instance creation are instructions that tell a system to destroy instances. If an object is destroyed, it is simply removed and not available anymore in the system's set of instances; if a component instance is destroyed, all of its inner objects are destroyed as well. Fig. 5.21 depicts the relations symbols for the destruction of instances.

Similar to the creation statements, we distinguish the symbols *ObjectDestruction* and *ComponentInstanceDestruction*. Both have in common, that they require a reference to the instance that should be destroyed, referred to by $destructionReference \subseteq InstanceDestruction \times ReferenceVariable$. Constraints related to the destruction are: first, component destruction statements can only refer to component instances; second, object destruction

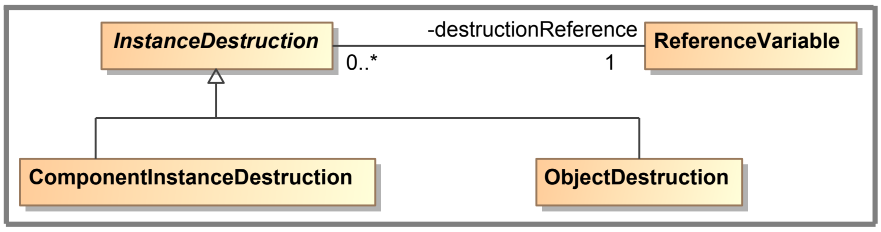


Figure 5.21: Relation symbols for instance destruction.

Relation symbol r	Interpreting relation r^S in the system S
StatementNode	$\{\ldots, 60, \ldots\}$
ObjectDestruction	$\{60\}$
destructionReference	$\{(60, 54)\}$

Table 5.8: Relations of the example from Tab. 5.7 are further extended by a object destruction statement.

statements can only refer to objects; third, null cannot be used as reference. Following axioms ensure this in Φ_{CBSD}:

$$\begin{aligned}\mathbf{CBSD_{48}} := \forall x \forall y \forall z : (&ComponentInstanceDestruction(x) \land Type(y) \land \\ &ReferenceVariable(z) \land destructionReference(x,z) \land \\ &hasType(z,y) \rightarrow Component(y))\end{aligned} \tag{5.57}$$

$$\begin{aligned}\mathbf{CBSD_{49}} := \forall x \forall y \forall z : (&ObjectDestruction(x) \land Type(y) \land \\ &ReferenceVariable(z) \land destructionReference(x,z) \land \\ &hasType(z,y) \rightarrow Class(y) \lor Interface(y))\end{aligned} \tag{5.58}$$

$$\mathbf{CBSD_{50}} := \forall x \forall y : (ObjectDestruction(x) \land destructionReference(x,y) \rightarrow y \neq null) \tag{5.59}$$

Let us assume, the following line of pseudo code follows the previous lines of the method body illustrated in this section so far:

```
delete_object w
```

where `delete_object` is the keyword for an object destruction statement. The relations of Tab. 5.7 could be extended as depicted in Tab. 5.8 to capture the additional statement (statement node with $id = 60$).

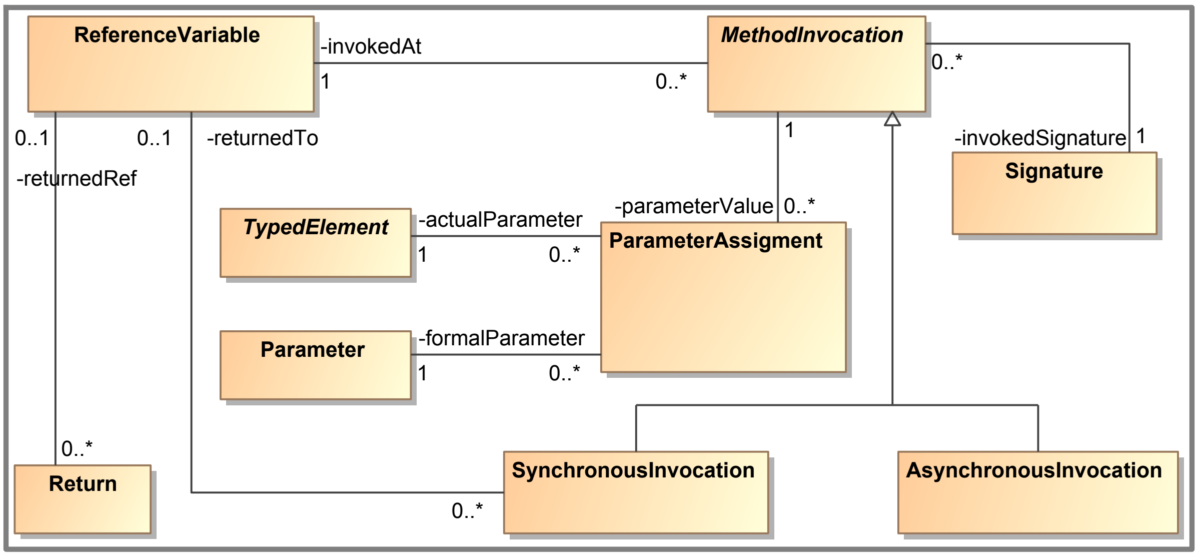


Figure 5.22: Method invocation statements

Method Invocation

There are two different forms of communication in component-based systems — synchronous and asynchronous sending of messages. Both imply that a method at the receiving instance is invoked. The sender of a synchronous message is blocked until the message is processed by the receiver and the answer is received by the sender. After that, the sender can proceed. In contrast, the sender of an asynchronous message drops the message and proceeds in its own control flow. Meanwhile, the message is waiting at the receiver to be processed. While the advantage of asynchronous messages is that the sender can continue its work, it is never ensured for an asynchronous message that it is processed or that a result will be sent to the original sender.

Fig. 5.22 depicts the relation symbols for method invocations. Two unary relation symbols represent the different forms of invocation, *SynchronousInvocation* and *AsynchronousInvocation*, respectively. The commonalities are covered by *MethodInvocation*. All method invocation, either synchronous or asynchronous, refer to a signature that is called at the receiver to process the message; this is modelled by $invokedSignature \subseteq MethodInvocation \times Signature$. The receiver object is identified by a reference variable pointing to the object — formally the invocation and that reference variable are related by the relation symbol $invokedAt \subseteq MethodInvocation \times ReferenceVariable$.

The class that serves as type of the reference variable has to provide a method body for the signature of the invocation. Otherwise, the invocation would not be possible. The following axiom of Φ_{CBSD} defines this constraint:

$$\begin{aligned}\mathbf{CBSD_{51}} := \forall x \forall y \forall z : (&MethodInvocation(x) \wedge Signature(y) \wedge \\ &ReferenceVariable(z) \wedge invokedSignature(x, y) \wedge \\ &invokedAt(x, z) \rightarrow (\exists t \exists t' : hasType(z, t) \wedge \\ &extends^{+}(t, t') \wedge (\exists m : definesMethod(t', m) \wedge \\ &hasSignature(m, y))))\end{aligned} \tag{5.60}$$

Furthermore, method invocations bind formal parameter to actual parameters. *ParameterBinding* reflects those bindings; its reference to *Parameter* by *formalParameter* $\subseteq$ *ParameterBinding* $\times$ *Parameter* refers to the formal parameter of a binding; *actualParameter* $\subseteq$ *ParameterBinding* $\times$ *TypedElement* refers to the actual parameter that could be another reference variable or some constant symbol. Each single parameter binding for a method invocation is related to the invocation by *boundParameter* $\subseteq$ *MethodInvocation* $\times$ *ParameterBinding*. Each formal parameter has of course to be a parameter that appears in the signature:

$$\begin{aligned}\mathbf{CBSD_{52}} := \forall x \forall y \forall z \forall z' : (&ParameterBinding(x) \wedge Parameter(y) \wedge \\ &MethodInvocation(z) \wedge Signature(z') \wedge \\ &formalParameter(x, y) \wedge boundParameter(z, x) \wedge \\ &invokedSignature(z, z') \rightarrow hasParameter(z', y))\end{aligned} \tag{5.61}$$

We furthermore claim that if a signature is invoked, each of the signature parameters (except the return parameter) appears in exactly one binding as formal parameter:

$$\begin{aligned}\mathbf{CBSD_{53}} := \forall x \forall y : (&MethodInvocation(x) \wedge Signature(y) \wedge \\ &invokedSignature(x, y) \rightarrow (\forall p : Parameter(p) \wedge hasParameter(y, p) \wedge \\ &\neg\, name(p, \text{“return”}) \rightarrow (!\exists b : ParameterBinding(b) \wedge \\ &boundParameter(x, b) \wedge formalParameter(b, p))))\end{aligned} \tag{5.62}$$

Synchronous invocations can optionally pass the result to a reference variable[7]. This is represented by *returnedTo* $\subseteq$ *SynchronousInvocation* $\times$ *ReferenceVariable*. It has to hold in addition that the reference variable addressed by *returnedTo* is the same as that one, to which the return parameter is bound by the corresponding binding:

$$\begin{aligned}\mathbf{CBSD_{54}} := \forall x \forall y \forall z : (&MethodInvocation(x) \wedge Signature(y) \wedge \\ &Parameter(z) \wedge invokedSignature(x, y) \wedge hasParameter(y, z) \wedge \\ &name(z, \text{“return”}) \rightarrow (\forall b \forall a : (boundParameter(x, b) \wedge \\ &formalParameter(b, z) \wedge actualParameter(b, a)) \leftrightarrow returnedTo(x, a)))\end{aligned} \tag{5.63}$$

When the receiving instance finishes the processing of a method invocation, it can return a result to the sender. This can be any typed element that is type-compatible to the declared return

[7]In this model, return statements for asynchronous invocations are ignored.

Relation symbol r	Interpreting relation r^S in the system S
StatementNode	$\{\ldots, 65, 70, \ldots\}$
LocalVariable	$\{\ldots, 68, \ldots\}$
ConstantSymbol	$\{66\}$ the constant “5”
AsynchronousInvocation	$\{65\}$
ParameterBinding	$\{67\}$
Return	$\{70\}$
invokedAt	$\{(60, 50)\}$
invokedSignature	$\{(60, 32)\}$
formalParameter	$\{(60, 33)\}$
actualParameter	$\{(60, 66)\}$
boundParameter	$\{(65, 67)\}$
returnedValue	$\{(70, 68)\}$

Table 5.9: Relations of the example from Tab. 5.8 are further extended by an asynchronous invocation and a return statement.

type of the method. The corresponding statement is modelled in τ_{CBSD} by the relation symbol *Return*. The returned element is represented by a typed element connected via $\mathit{returnedValue} \subseteq \mathit{Return} \times \mathit{TypedElement}$.

Consider the following example. The exemplary method body of this section contains an asynchronous invocation of the method c at the local variable v; the integer parameter I is bound to the constant value “5”:

```
v.c(5)
```

Furthermore, a local variable ($id = 68$) with the declared type $I2$ is defined and returned after some omitted operations. Table 5.9 depicts the additional relations that reflect those two additional statements. The statement with $id = 65$ stands for the invocation. Its parameter binding ($id = 67$) binds the constant “5” ($id = 66$) to formal parameter I (see Fig. 5.11, p. 85). The method is called at v which can be seen by the fact that $(60, 50) \in \mathit{invokedAt}^S$. The return statement ($id = 70$) return the new local variable ($id = 68$) — correctly represented by $(70, 68) \in \mathit{returnedValue}^S$.

5.4. Abstraction Classification of Models and Model Compliance

This section introduces the abstraction classes of models and relates them to the classification common in model-driven development. The term of compliance between models is defined based on this classification.

5.4.1. Model-Classification in Model-Driven Development

In model-driven development, models are distinguished into platform-independent and platform-specific models, as explained in Sec. 2.2.2. Although illustrations of frameworks like MDA suggest a strict, global order of platform-independent and -specific models, the classification is relative to the platform. For example, a Java-specific model can be a platform-independent model for some component technology like EJB. Nevertheless, a platform separates the set of all possible models into those that describe systems using the platform and those that do not. This separation is basis for frameworks like MDA and the model-driven development process as described in Sec. 2.2.2.

As experiences in model-driven development show [BLW05, BE08, MRA05, Sta06], platform specialization is useful, since it allows building systems more efficiently by providing automatic model and code generation — although results of experiences must be observed carefully [MD08] . It shifts the focus from developing technology-dependent models on creating more durable conceptual models. To enable automatic model transformation and code generation, transformation rules are defined. They describe how elements of source models are transformed into elements of a target models.

Hence, this separation supports developers — architects, designers, and programmers — to switch platforms because they are able to model systems independently of certain platforms. Such a model does not describe what should be content of architectural, design, and implementation models, or who should be responsible for modelling certain aspects of the system.

Those aspects are covered by the three different classes of strategic, tactical, and implementation statement as defined in Sec. 5.2. Models can be classified by the class of the statement they make about the system. Intuitively, the three classes distinguish models by the broadness of their impact to the overall system and how it may evolve. Consider, for example, the natural language statement: “A is a component, and B is an interface”. This is clearly an implementation statement and has a very limited impact; other elements than A and B can be deleted, new elements can be added, and the statement will stay true. The statement “Component A is an observer of C”, a design statement, has the impact that not every element can be deleted. For example, the statement implies that C provides a method for notification, which cannot be removed. Finally, statements like “No cyclic dependencies are allowed!”, a strategic statement, have the greatest impact to the way the system may change; as a global constraint, it can be violated by any addition made to the system.

The three classes can be easily mapped to the different developing steps as described in Sec. 2.2. Constraints that need to be checked after every change of a system, i.e. strategic statements, can surely be called “fundamental” and should be defined by software architects. On the other hand, models or code that define no additional constraints but only facts, like implementation statements do, are those that programmers normally construct. In between, there is the software designer who rather decides on local design solutions and constraints than on fundamental structures - exactly what tactical statements do.

Both classification criteria are orthogonal to each other. The first criterion, platform-specialization, distinguishes models by technologies, programming or modelling languages. The second criterion takes into account the system impact of design statements and who should

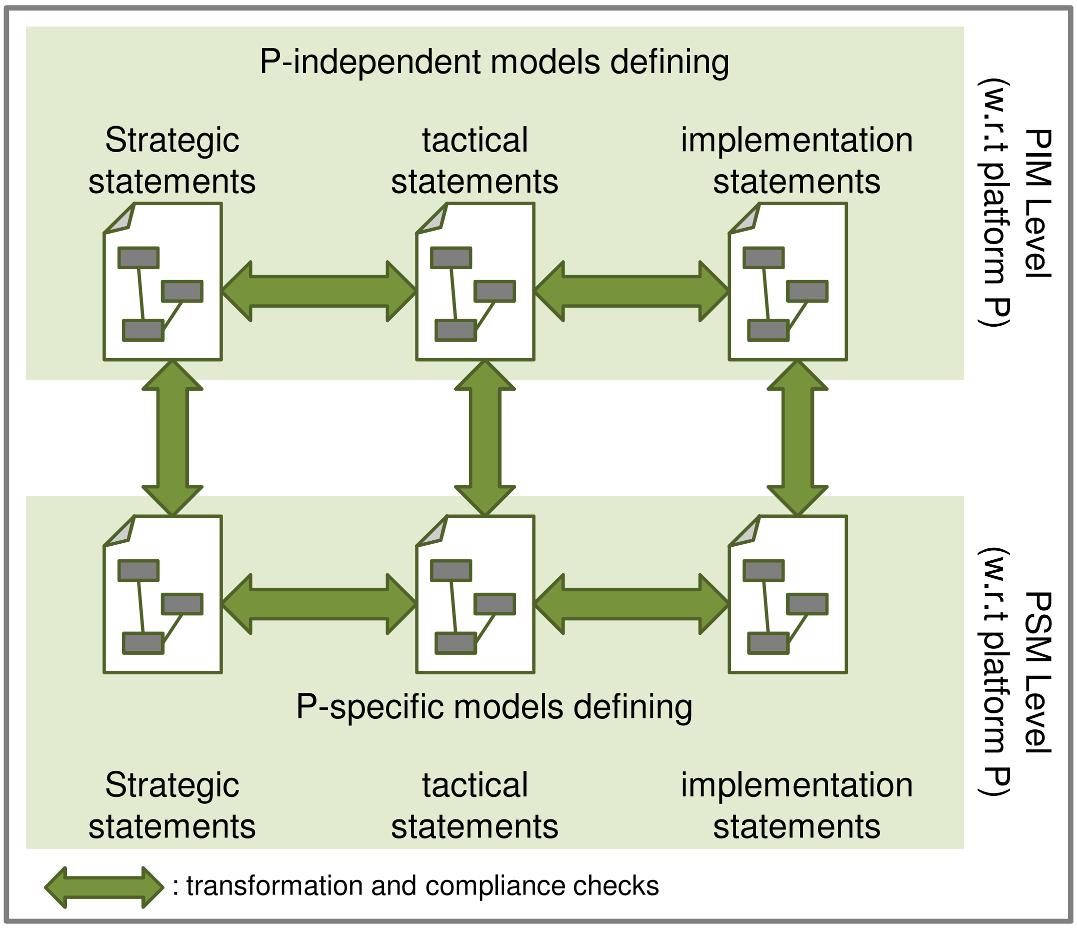

P-independent models defining
Strategic statements
tactical statements
implementation statements
PIM Level (w.r.t platform P)
P-specific models defining
Strategic statements
tactical statements
implementation statements
PSM Level (w.r.t platform P)
: transformation and compliance checks

Figure 5.23: Orthogonal model abstraction levels.

be responsible for them.

The combination of both criteria leads to refined illustration of model-driven development processes, as depicted in Fig. 5.23. For each level of platform specialization, there can be models of the three different classes strategic, tactical, or implementation, named accordingly to the statements that they make about the system. Models on a platform-independent level (w.r.t platform P) do not contain elements specific to that platform.

This may seem unusual at first because this means that there can be models for platforms very closely related to the implementation of a system (e.g., a programming language) that should be developed by software architects. But consider a statement that requires all classes to have a common base class for the C++ implementation of the system [EK03]. This strategic statement should of course not be made by a single programmer, but by a software architect. On the other extreme, the statement “A is a component” is an implementation statement but on a very abstract platform because the component technology (a platform) is not defined yet.

Fig. 5.23 also depicts model transformations as important building blocks of model-driven development. They transform platform-independent into platform-specific models base on transformation definitions. Model transformations can also be executed to transform models of greater evolutionary impact into models of less greater impact class. For example, a transformation rule could create corresponding packages of a Java implementation for the architectural layers, defined in a strategic statement, of a system.

One goal of using model transformation is to achieve a higher automation. This goal is achieved to a higher degree, the more complete the generated target model is, and the less of the model has to be created by hand. With respect to that, it is important to note that highly intensional models, that mainly make constraint about the system, are in general less adequate as source models for transformations, as discussed in Sec. 2.2.2. Hence, instead of compliance by construction through transformation, compliance checking is needed as well.

In the following, we will focus on the definition of compliance between models on the same level of platform specialization but with different level of abstraction regarding their class of statements.

5.4.2. Definitions for the Formalization of Models

In the following, the term of a model and related concepts will be defined. Examples of models of different abstraction levels will be given.

Models

Models provide in general a simplified view on some subject existing in reality. It is simplified in the sense that the model contains only information relevant for the purpose of usage and omits the full complexity of the subject.

In this work, we are interested in models that provide abstracting descriptions of component-based systems. As always in model-based development, models should additionally have a defined syntax. This leads to the following definition:

Definition 14 (Model of a component-based system, model classes) A *model M* of a component-based system is a description of a component-based system and has the following properties:

- M has a defined abstract syntax
- M defines a set of logical statements $\Phi^M = \Phi^M_{ext} \dot{\cup} \Phi^M_{int}$ whereas
  - each $\varphi \in \Phi^M$ is a τ_M-formula, $\tau_M \subseteq \tau_{CBSD}$,
  - Φ^M_{ext} contains only extensional statements, and
  - Φ^M_{int} contains only intensional statements.

  We furthermore call $r \in \Phi^M_{int}$ a *design rule* if it is a local statement; it is called an *architectural rule* if it is an non-local statement.

A system $s \in \mathcal{K}_{CBSD}$ *conforms* to a model M if and only if $s \models \Phi^M$.

Each model M is in exactly one class of models that corresponds to the classes of statements the model defines:

- A model M is an *implementation model* if and only if $\Phi^M_{int} = \emptyset$.

- A model M is a *design model* if and only if Φ^M_{int} contains only design rules.
- A model M is an *architectural model* if and only if Φ^M_{int} contains architectural rules.

All references of "intensionality" or "extensionality" are meant w.r.t. $\tau_{CBSD,cl}$, the set of close relations from τ_{CBSD} (see Def. 16).

This definition does not restrict the term "model" to instances of meta models as common and defined in model-driven development. According to this definition, also grammars for textual languages or schema definitions common for XML documents can serve as syntax definitions.

Furthermore, we define the set of satisfying structures of a model as follows:

Definition 15 (Satisfying and minimal structures/systems for models) The set $\mathcal{F}^M$ of satisfying structures for a model M (as an artefact in the model-driven engineering sense) is defined as

$$\mathcal{F}^M := \{s \mid s \models \Phi^M\}$$

whereas s are τ-structures with $\tau \supseteq \tau_{CBSD}$.

The set $\mathcal{S}^M$ of of satisfying systems for a model M (as an artefact in the model-driven engineering sense) is defined as

$$\mathcal{S}^M := \{s \in \mathcal{K}_{CBSD} \mid s \models \Phi^M\}$$

The set of minimal satisfying structures for M is defined as

$$\mathcal{F}^M_\perp := \{\text{s is minimal structure for } \Phi^M\}$$

Furthermore, we define the set of minimal models for M as

$$\mathcal{S}^M_\perp := \{s \in \mathcal{S}^M \mid \text{ there is no substructure } s' \text{ of } s \text{ with } s' \in \mathcal{S}^M\}$$

The definition furthermore states that a statement's property of being intensional or extensional is determined w.r.t $\tau_{CBSD,cl}$. This means that adding tuples from those relations constitute a modification of a structure according to the definition of modifying and additive extensions (see Sec. 5.2).

Definition 16 (Close relations of τ_{CBSD}) The close relations $\tau_{CBSD,cl}$ of τ_{CBSD} are defined as

$$\begin{aligned}\tau_{CBSD,cl} :=\{&hasMember, definesSignature, definesMethod, hasParameter\\ &assignedTo, returnedValue, returnedTo\}\end{aligned}$$

Hence, the addition of tuples is interpreted as modification if it reflects one of the following system modifications:

- The addition adds either a member to an existing classifier, a method to an existing class, or a signature to an existing interface.

- The addition adds parameters to an existing signature.
- The addition changes an instance creation statement without a reference, that the new instance is assigned to, by adding such a reference.
- A return statement without a returned reference variable is changed by adding a corresponding reference variable.
- A synchronous invocation statement, that does not pass the result to any reference variable, is changed into an statement passing the result.

Concluding, this means that classifiers and interface are considered to be structural atomic units and single statements nodes to be behavioural atomic units by definition. Adding tuples of relations from $\tau_{CBSD,cl}$, in which the first component is an existing entity, is interpreted as modifying an atomic unit.

It is important for the classification of models to be aware of the atomic units and $\tau_{CBSD,cl}$. They basically define which sets of extensions we ignore when classifying a statement as local or non-local. Consider, for example, a statement which states that in a concrete instantiation of the observer pattern, the observer must provide an update method without parameters. Intuitively, this should be a tactical statement. Once, there is a valid instantiation of the pattern (described in a model) with a corresponding update method, the instantiation cannot become invalid by defining new instantiations, adding components, classes, etc. But formally, adding a parameter entity to the existing signature entity by *hasParameter* would be an addition that makes the pattern instance invalid if we would not take care of the close relations. $\tau_{CBSD,cl}$ hence defines that additions to relations in it can modify existing atomic units.

Implementation Models

According to the definition above, implementation models formulate only extensional statements about component-based systems. There are no intensional constraints defined by an implementation model of a system.

Recapitulating the definitions of Sec. 5.2, this means that a system satisfying an implementation model stays a satisfying system if it is extended or changed by removing entities that are not explicitly mentioned in the implementation model.

Trivial examples of extensional statements are tautologies. Tautologies are satisfied in every structure of the considered signature; they are hence satisfied in every extension and modification of a satisfying structure. Also models that repeat, for whatever reason, as sole statements axioms from Φ_{CBSD} or implications of them, postulate extensional statements — the axioms are true in every component-based systems, hence in every extension and modification of a component-based system.

However, models stating tautologies or axioms are uncommon. In most cases, implementation models make statements comparable to the informal example of Fig. 5.2; they list the instances of concepts, i.e. the extensions of concepts. Formally, this means that Φ^M_{ext} for an extensional model normally consists of formulae of the form $R(c_1, \ldots, c_n)$ whereas R is an n-ary relation symbol and the c_i are constant symbols, referring to single entities. Every system providing these extensions, or supersets of them, is a valid system.

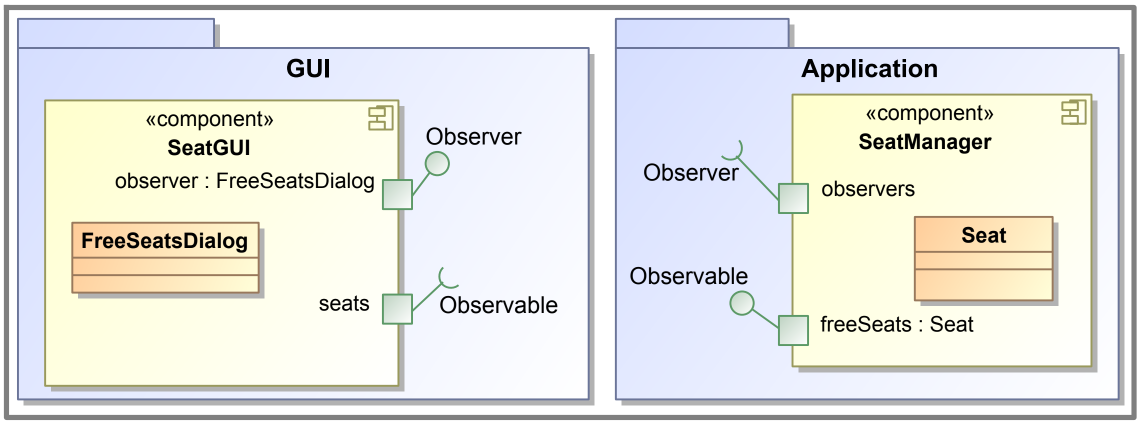


Figure 5.24: Exemplary implementation Model M_I — component specification.

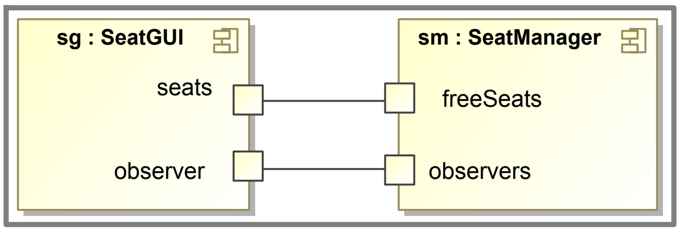


Figure 5.25: Exemplary implementation Model M_I — system configuration.

Fig. 5.24 depicts an exemplary UML component diagram. Note, that this and the following UML diagrams do not represent component-based systems as Fig. 5.9–5.17 do, but models of such systems. Hence, the diagram annotation «system model» is missing. It shows a cut-out of a seat reservation system consisting of a GUI component and a managing component which provides the functionality for making, modifying and removing reservations. The components are realized with help of the observer pattern, the GUI being an observer of the reservation manager component; whenever the set of reservations changes, every registered instance of the GUI is notified of the modification and can react, for example, by displaying a newly freed seat. The interfaces are implemented by classes inside the components. Fig. 5.25 shows in addition a composite structure diagram that depicts a system configuration which connects two instances of the components to form the system.

If we consider the two diagrams being views onto the same implementation model M_I, a possible set of statements Φ^{M_I} could be[8]:

$$
\begin{aligned}
\Phi^{M_I}_{int} =&\emptyset\\
\Phi^{M_I}_{ext} =&\{Package(Gui), Package(Application),\\
&Component(SeatGUI), Component(SeatManager),\\
&Interface(Observer), Interface(Observable),\\
&Class(FreeSeatsDialog), Class(Seat),\\
&ProvidedPort(observer), RequiredPort(seats),\\
&RequiredPort(observers), ProvidedPort(freeSeats),\\
&Part(sg), Part(sm), SystemConfiguration(s),\\
&containsComponent(Gui, SeatGUI),\\
&containsComponent(Application, SeatManager),\\
&containsInterface(Application, Observer),\\
&containsInterface(Application, Observable),\\
&providesInterface(SeatGUI, Observer), providesInterface(SeatManager,\\
&Observable), requiresInterface(SeatGUI, Observable),\\
&requiresInterface(SeatManager, Observer),\\
&containsClass(SeatGUI, FreeSeatDialog), containsClass(SeatManager, Seat),\\
&implements(FreeSeatsDialog, Observer), implements(Seat, Observable),\\
&encapsulates(SeatGUI, observer), encapsulates(SeatGUI, seats),\\
&encapsulates(SeatManager, observers), encapsulates(SeatManager, freeSeats),\\
&providedInterface(observer, Observer), providedInterface(freeSeats,\\
&Observable), requiredInterface(seats, Observable), requiredInterface(\\
&observers, Observer), hasType(observer, FreeSeatsDialog),\\
&hasType(freeSeats, Seat),\\
&configurationPart(s, sg), configurationPart(s, sm),\\
&configurationPart(s, con1), configurationPart(s, con2),\\
&hasType(sg, SeatGUI), hasType(sm, SeatManager),\\
&connectorSource(con1, seats), connectorSource(con2, observers),\\
&connectorTarget(con1, freeSeats), connectorTarget(con2, observer),\\
&sourceContext(con1, sg), sourceContext(con2, sm),\\
&targetContext(con1, sm), targetContext(con2, sg)\}
\end{aligned}
$$

The statements basically lists the elements of the relations that need to be present in a conforming system, for example, there have to be at least two components which are the entities with $id = SeatGUI$ and $id = SeatManager$.

[8]We use here the names of the model elements as identifiers for simplification, instead of numbered entities and constant symbols mapping to them.

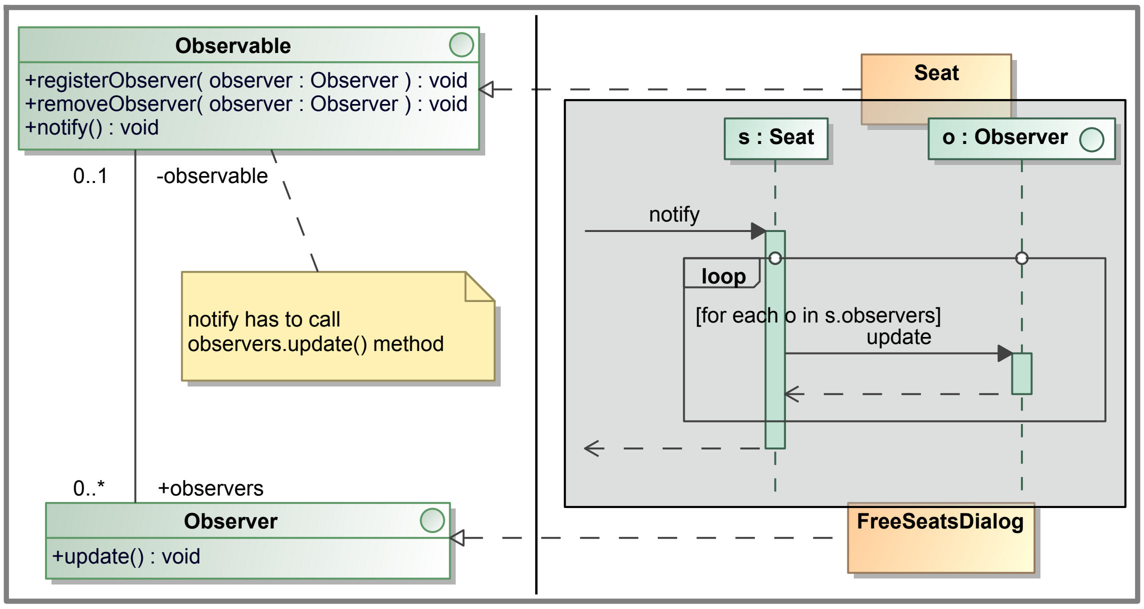

Observable
+registerObserver(observer : Observer) : void
+removeObserver(observer : Observer) : void
+notify() : void
0..1
-observable
notify has to call observers.update() method
0..*
+observers
Observer
+update() : void
Seat
s : Seat
o : Observer
notify
loop
[for each o in s.observers]
update
FreeSeatsDialog

Figure 5.26: Sketch of an observer pattern description.

Sets of extensional statements of the form like $\Phi_{ext}^{M_I}$ have the property that they have a unique minimal structure satisfying the set; each statement refers to exactly one relation tuple, thus the structure defining exactly these tuples is minimal. This property of this special form of extensional statements will be exploited for the operationalization of compliance checks (see Sec. 5.5).

Design Models

Loosely spoken, implementation models make “matter of fact” statements about systems in the sense that they claim the existence of elements and relationships. Design models, in addition, can postulate constraints that can be locally fulfilled. This means, once the constraint is satisfied, the system can be extended in any way — the constraints will hold. Or to describe it from the opposite: if an tactical statement of the model is violated, the system contains a local error.

Following Eden [EHK06], examples of statements, that design models can make, are:

- Design Patterns
- Programming Language Idioms
- Refactorings

Let us assume, for the exemplary seat reservation system exists a description how the observer pattern works. In most cases this might be a textual description from a pattern catalogue like [GHJV95] complemented by a UML diagram like in the left part of Fig. 5.26, describing the interfaces that concrete observers and observables have to implement. Since UML class diagrams are in general extensional, as discussed in [EHK06], there are some (informal) intensional constraints, like the one stating that the notification has to call the update methods of the observers. This might be expressed as an τ_{CBSD}-expression as follows:

$$\begin{aligned} observable\_calls\_upd(o) := {} & implements^{+}(o, Observable) \rightarrow \\ & \exists m \exists i \exists s : definesMethod(o, m) \wedge \\ & name(m, \text{"notify"}) \wedge containsStatement(m, i) \wedge \\ & invokedSignature(i, s) \wedge \\ & name(s, \text{"update"}) \wedge invokedAt(i, observers) \end{aligned}$$

If the class Seat is a proper implementation of Observable the statement $observable\_calls\_update(Seat)$ will be satisfied. Of course, this would be only a partial descriptions of the constraints the observer patterns defines.

Hence, let us assume that the diagrams in Fig. 5.26, the depicted class diagram, and the embedded sequence diagram, describe another model M_D of the seat reservation system. Then the corresponding sets of statements could be, omitting the details for the definition of signatures and methods for registering and removing observers, as well as the full control flow graph for notify:

$$\begin{aligned} \Phi_{ext}^{M_D} = \{ & Interface(Observer), Interface(Observable), \\ & Class(FreeSeatsDialog), Class(Seat), \\ & implements(FreeSeatsDialog, Observer), \\ & implements(Seat, Observable), \\ & Member(observers), hasMember(Observable, observers), \\ & definesSignature(Observer, update_S), \\ & definesSignature(Observable, notify_S), \\ & definesMethod(FreeSeatDialog, update_M), \\ & hasSignature(update_M, update_S), \\ & definesMethod(Seat, notify_M), \\ & hasSignature(notify_M, notify_S), \\ & \ldots, MethodInvocation(upd), \\ & containsStatement(notify_M, upd), \\ & invokedSignature(upd, update_S), \\ & invokedAt(upd, observers)\} \cup \Phi_{ext}^{M_I} \\ \Phi_{int}^{M_D} = \{ & observable\_calls\_upd(Seat)\} \end{aligned}$$

In contrast to the implementation models mentioned in the last section, design models do not have a unique minimal satisfying structure in general. Their intensional design statements usually contain free or bound variables. If a variable is bound to an existential quantifier, for example, it is in general possible that more than one binding will lead to a minimal satisfying

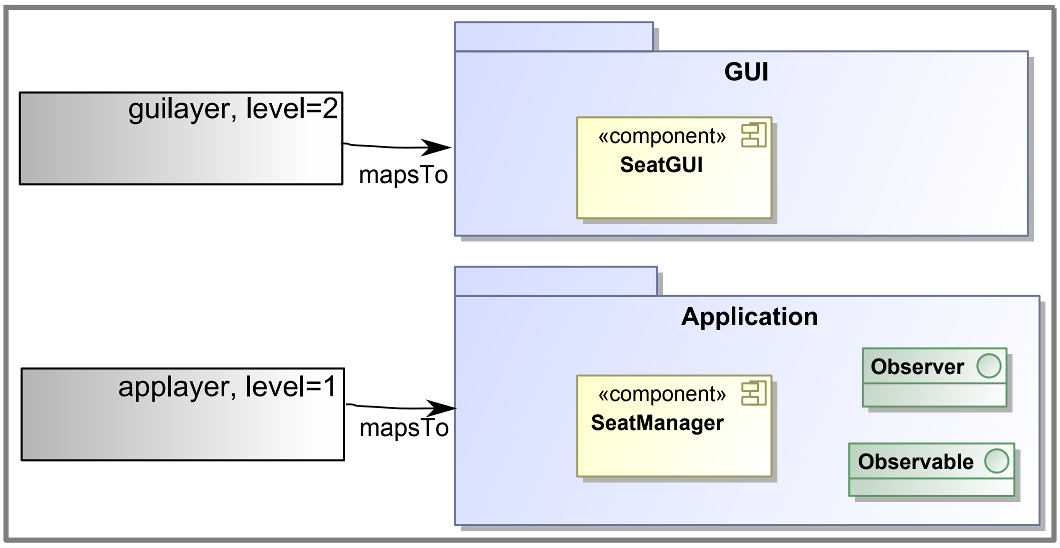


Figure 5.27: An architectural model introducing layers.

structure of the statement. In the statement above, for example, an arbitrary entity reflecting the notify method could make the existentially quantified part true; hence, there are infinitely many minimal satisfying structures.

Architectural Models

Architectural models, in contrast to design models, define non-local constraints. This means that a system satisfying an architectural model can always be extended in a way that it forms a system that does not satisfy the model. Examples for architectural models are, as describes in [EHK06]:

- Architectural patterns and styles.
- Design principles and paradigms, like information hiding or the main principles of object-orientation.
- Implementation guidelines.

Let us consider the layers pattern, broadly recognized as architectural pattern [BMRS96]. Layers organize the parts of systems hierarchically, and group components at different levels. Components are only allowed to access components at the same level or a level below[9].

A common reference architecture for information systems is build upon the layers pattern, consisting of a user interface layer, the application layer and a persistence layer, as introduced in Sec. 4.1.2. Let us assume, our seat reservation system follows this approach as depicted in Fig. 5.27. We could assume, that layers are mapped to packages of a system and can be formalized in an architectural model as follows:

[9]We will define the layer pattern and the corresponding logical statements in much more detail in Sec. 6.2.1.

$$\begin{aligned}\Phi_{ext}^{M_A} =& \{Layer(guilayer), Layer(applayer),\\ & level(guilayer, 2), level(applayer, 1),\\ & mapsToPkg(guilayer, Gui), mapsToPkg(applayer, Application)\}\\ & \cup \Phi_{ext}^{M_I}\\ \Phi_{int}^{M_A} =& \{legalDep(guilayer), legalDep(applayer)\}\end{aligned}$$

legalDep is a constraint that defines for a layer that all dependencies, that exist because of components requiring or providing interfaces, stay in a layer, or are directed towards a layer below:

$$\begin{aligned}legalDep(l) :=& \forall p \forall c \forall i : mapsToPkg(l, p) \wedge containsComponent(p, c) \wedge\\ & (requiresInterface(c, i) \vee providedInterface(c, i)) \rightarrow\\ & (\exists l' \exists p' : mapsToPkg(l', p') \wedge containsInterface(p', i) \wedge\\ & \forall x \forall x' : (level(l, x) \wedge level(l', x') \rightarrow x \leq x'))\end{aligned}$$

The minimal model for M^I is also a model for M^A, as can obviously be seen; both interfaces are defined in the package mapped to the application layer, hence all dependencies go from towards the lowest layer available. Thus, all dependencies conform to the layer levels. But such a system can be easily be extended to a non-satisfying system by adding layers below the application layer (such as a persistence layer) which contain components that require an observer or an observable interface.

As for design models, it cannot be assumed that an architectural model has a unique minimal satisfying structure.

Refinement and Compliance

To define refinement and compliance between models we have to investigate relations between their sets of structures that satisfy their statements.

Refinement is often defined by saying that every system that conforms to the refining model is also conform to the refined model (for example, see [Rau04]), or formally $sem(M_{refining}) \subseteq sem(M_{refined})$ whereas $sem(M)$ denotes the semantics of a model M in terms of systems conforming to M.

However, in the proposed approach it is possible that, e.g. an architectural model contains concepts, like layers, that are not available in an implementation or design model. The common core of concepts is defined by the relation symbols in τ_{CBSD} but additional concepts can be added. Hence, the τ_{CBSD}-reductions (see Sec. 5.2) have to be considered for refinement:

Definition 17 (Refinement between models) Let M and M' be two models. M is a refinement of M' if and only if for each $s \in \mathcal{S}^M$ there is a $s' \in \mathcal{S}^{M'}$ such that $s \restriction \tau_{CBSD} = s' \restriction \tau_{CBSD}$.

The refinement relationship between models is very strong. Intuitively spoken, every system s for a model M is also a system for M' if M is a refinement for M'. It is obvious that this constraint is too strong for the relationship between design or implementation models and architectural models. Both kinds of models, design and implementation models, describe sets of satisfying systems that are closed under addition of entities and tuples, which results from the definition of locality and extensionality. Since this does not hold for non-local models in general, the relationship between architectural models and less abstract models cannot be described as refinement in general.

The desired relation of compliance has hence to be weaker than refinement. The minimal satisfying systems of a model can be interpreted as those systems which are completely described by the model; larger systems supersetting a minimal system contain elements that are not explicitly required by the model and that are not described by it. If it is because of one of these elements in a structure for a design model, that the system violates the constraints of an architectural model, it is strange to say the design model is not compliant with the architectural model. The need for that element is not expressed in the design model.

Instead, compliance of a model M with a model M' should express that a minimal system satisfying M should not contradict the constraints of M'. This means that each such minimal system could be added to minimal system of M' resulting in a satisfying system for M'. More formally, we state:

Definition 18 (Compliance between models) Let M and M' be two models whereas M uses the signature $\tau_{CBSD} \cup \tau_M$, M' uses $\tau_{CBSD} \cup \tau_{M'}$, and $\tau_M \cap \tau_{M'} = \emptyset$. M is compliant with M' if and only if the following holds: for every $s \in \mathcal{S}^M_\perp$ there is an $s' \in \mathcal{S}^{M'}_\perp$ such that for t, which consists of the union of s and those entities and tuples of s', that are elements of relations defined by $\tau_{M'}$, holds: $t \in \mathcal{S}^{M'}$.

Intuitively spoken, this definition of compliance says that M can be extended by the information contained in M' in a manner that it is conform to M'. By restricting the extension to relations that are not defined in τ_{CBSD}, it is ensured that if M' makes extensional statements about component-based concepts (“C is a component”) are consistently modelled in M'.

Of special interest with regard to the task of architectural compliance checking are two cases of compliance:

1. Compliance of implementation models with design models. This issue arises, for example, if the implementation has to be checked for compliance to tactical statements like design patterns.

2. Compliance with architectural models. This issue arises, if architectural models, constituting strategic statements, have to be checked in the design and implementation of a system.

The two cases as will be explained and discussed in the following sections.

5.4.3. Compliance of Implementation Models towards Design Models

The practical main use case, in which extensional models are checked for compliance with a design model, is that the implementation of a system given as source code is checked against a design model written in UML, or any other modelling language. The definition of compliance ensures that everything that the design tells about specific component, interfaces, and relationships, is also implemented by source code. Because of the fact that only design-specific concepts can be added to the systems that satisfy the logical statements defined by the code to check if also the design model is satisfied, every component and every other concept from τ_{CBSD} modelled in the design model has also to appear in code.

The implementation model example of Sec. 5.4.2 can now be checked for compliance with the design model of the same section. The implementation model M_I has a unique satisfying structure which interprets the relation symbols by the minimal sets of tuples required for Φ^{M_I}, for example, $Component = \{SeatGUI, SeatManager\}$ or $containsClass = \{(SeatGUI, FreeSeatsDialog), (SeatManager, Seat)\}$. The definition above states that this structure can be extended by a minimal system for M_D if M_I is compliant with M_D. It can easily be seen that this is not possible, since M_D requires the existence of *MethodBody* tuples, for example for the update method, that are not contained in the satisfying structure for M_I. They may not be added to the system because *MethodBody* is defined in τ_{CBSD}, hence there cannot exist a minimal system for M_D from which we could take the additional elements according to the definition of compliance. M_I does not specify the update method and can thus not be compliant with the design model by definition.

Let us assume, that the diagram of Fig. 5.24 is modified by adding the necessary signatures to the observer and observable interface, as depicted in the design model of Fig. 5.26, i.e. update and notify signatures. Furthermore, the corresponding methods are implemented in a way that the notification calls the update methods. Note, that the source code of that method bodies will include many more statements, more than those specified as required in the design model.

In this case, the source code, i.e. the implementation model, is compliant with the design model. A minimal system for the implementation code would now contain corresponding method body entities, statement node entities, and according relationships. If observer related tuples are added from a minimal system for M_D. This leads to a system that contains the tuples reflecting these additional facts — it is basically a structure which also satisfies M_D. Consequently, the modified implementation model is compliant with M_D, since a set of design-specific tuples, the empty set, can be added to from a structure satisfying M_D.

As an interesting aspect, the compliance of an implementation model with a design model, also implies the refinement between both, whereas the implementation model refines the design model. Let us temporarily assume, that the considered implementation and design models do not use additional relation symbols to that of τ_{CBSD}. According to the definition above, every minimal satisfying system of an implementation model M_I is also a model of compliant design models. For both kind of models, we know by the definition of extensional and local statements, that every addition to a satisfying structure leads to another satisfying structure. Hence, if every minimal model of M_I is also a satisfying structure of the design model M_D, every larger model for M_I does satisfy the statements of M_D, too. Hence, every satisfying system satisfies M_D and M_I is a refinement of M_D, too.

If M_I and M_D also use other relation symbols than those defined in τ_{CBSD}, the reduction to τ_{CBSD} has to be observed. If M_I uses relation symbols that are not known in M_D, the case is simple — the intensional constraints cannot use them, and hence, corresponding tuples are irrelevant for the evaluation of the logical statements. In this case, M_I is a refinement of M_D, too. If M_D uses such relation symbols, they can be important for the evaluation because they are used as terms in the statements. However, these statements are local, thus adding tuples to a satisfying system is safe. Hence, the same tuples could be added to a minimal system of M_D, that are inserted to form an arbitrary satisfying system for M_I from a minimal system for M_I. Consequently, the refinement relationship holds also in this case.

5.4.4. Compliance towards Architectural Models

By definition, a component-based system satisfying an architectural model can be modified in a way that it does not conform to the model anymore. Consider a system, that is layered according to an architectural model; components can be added to the system that require and provide interfaces from layers which they are not allowed to access. This shows that compliance of an arbitrary model with an architectural model does not imply refinement in contrast to the observations regarding compliance with a design model.

Let us consider the examples M_D and M_A of the previous section. The minimal satisfying systems for M_D contain exactly the components and classes required for the extensional statements of the model, and the update methods provides an implementation for the update method as it would be expected according to the pattern [GHJV95]. The details of the implementation are not relevant with regard to the question if M_D is compliant with M_A because the statements in M_A do not refer to any relation symbols used to model method bodies.

Since the minimal models contain exactly the component and interfaces that correspond to those in Fig.5.27, it is obvious that they can be extended to form a model for M_A. All that has to be done is to add corresponding *Layer*, *level*, and *mapsToPkg* tuples to such an minimal structure. Hence, if s is a minimal model for M_D the following structure s' is a model for M_A:

$$\begin{aligned}
Layer^{s'} &= Layer^{s} \cup \{guilayer, applayer\} \\
level^{s'} &= level^{s} \cup \{(guilayer, 2), (applayer, 1)\} \\
mapsToPkg^{s'} &= mapsToPkg^{s} \cup \{(guilayer, Gui), \\
&\qquad (applayer, Application)\} \\
R^{s'} &= R^{s} \text{ for all other relation symbols}
\end{aligned}$$

$Layer^s$, $level^s$, and $mapsToPkg^s$ are empty sets. The addition to form s' corresponds to the requirements of Def. 18, and s' is a model of M_A. Thus, M_D is compliant with M_A

5.5. Operationalization of Compliance Checking

This chapter is about the execution of compliance checks based on the formal definitions in the subsequent sections. The execution of compliance checks is separated into different steps.

First, models have to be transformed into logical expressions that represent the statements contained in a model about systems. The definition of transformations is provided at the meta-model level, every model instance is transformed according to that language-specific rules. This aspect is covered in Sec. 5.5.1.

In a second step, the minimal satisfying systems for models are constructed. This is only possible if certain assumptions can be made about the models that are going to be checked. Section 5.5.2 describes those assumptions and the generation of minimal satisfying systems in cases when the assumptions are true.

Finally, the compliance check can be executed according to the definition of compliance. A merged system is constructed from the minimal systems for the participating models and serves as input for model checking. In Sec.5.5.3, a compliance checking algorithm will be introduced.

5.5.1. Transforming Models into Logical Statements

To transform models into logical statements, we have to define the transformation rules. They express for a certain kind of models, i.e. models written in a defined language like UML, which logical statements are generated for a model. This means that transformation rules have to refer to elements of the meta model, for example, to express that every instance of the UML meta model concept Component results in the expression $Component(e_{this})$ whereas *this* refers to the id of the instance.

Given such transformation rules, a model can be traversed and transformed, model element by model element. For example, a UML model is traversed, and if a component is visited, it is transformed according to the rule above. Other kinds of model elements are transformed according to different rules. The union of the generated statements for the model elements of a model M is the set Φ^M of logical statements that have to be satisfied in conforming component-based systems.

The way transformations are defined and executed should be language independent, and hence must be independent from a specific meta model. Thus, it is helpful to assume that all meta models are defined using the same concepts. This way we can easily define the transformation rules as function mapping instances of those concepts to logical expressions over τ_{CBSD}. Additionally, an algorithmic way to generate those statements language-independently can be defined.

In the following, meta models are considered to be instances of the *EMOF* meta-meta model which is a subset of the MOF approach defined by the OMG [Obj06]. In this approach, model-driven development is separated into three different abstraction levels for artefacts called M_0, M_1, M_2, and M_3. On M_0, subjects of observations are concrete systems. Models of such systems are on M_1. The languages, in which models are written, are defined by meta models; they are on M_2. Finally, M_3 defines the common meta-meta model for MOF; it describes the languages in which meta models are written.

Figure 5.28 depicts in the topmost part the cut-out of the EMOF meta meta model which is of special interest for the proposed transformation into logical expressions. EMOF defines as main concepts classes of which specific meta model are constructed. Classes can define properties typed with classes or other specializations of Type as depicted. This means, that in EMOF meta models mainly consist of meta classes with meta attributes that refer to other classes (or primitive types). For example, the UML meta model defines the meta class Interface as an instance of the meta-meta class Class. Interface has a (meta) property called ownedAttribute which is an instance of the meta-meta class Property.

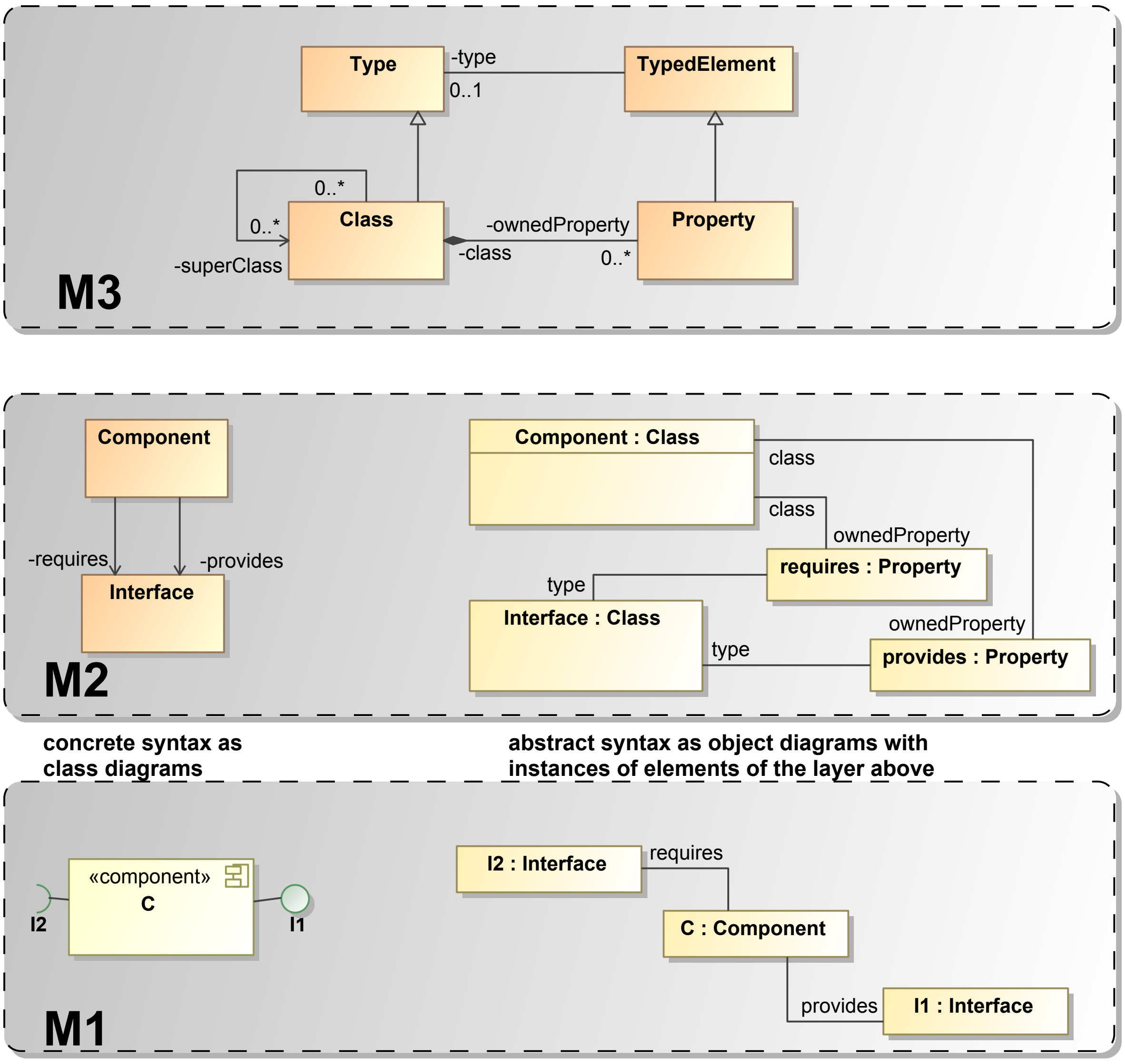


Figure 5.28: UML in the layers of the MOF.

Taking this structure into account, we can now define that a transformation definition maps an instance of Class (from M_3) to some extensional and intensional expressions, depending on the language or meta model L and the signature $\tau_L \supseteq \tau_{CBSD}$:

$$T^L : Class^L \rightarrow \mathcal{P}(EXT^{\tau_L}) \times \mathcal{P}(INT^{\tau_L})$$

$Class^L$ denotes the set of instances of Class (from level M_3) in L. EXT^{τ_L} and INT^{τ_L}, respectively, denote the set of extensional and intensional statements over τ_L.

All expression can use the variable *this* which is replaced during the execution of the rule by the identifier of the model element to which the rule is applied. Furthermore, the "*this.property*" whereas *property* denotes an inherited or owned property. The variable is replaced by the element the property refers to by its type. If the multiplicity is greater than one, a statement for each possible replacement is generated. This navigation works also iteratively; *this.x.y* determines the property *y* of the element which is referenced by *this.x*.

Figure 5.28 depicts a cut-out of the UML meta model and a model inside the M-layers of the (E)MOF. It shows the elements required to model the provides and requires relationships between interfaces and components. The corresponding classes are Component and Interface. Component has two properties, one to refer to the provided, one to the required interfaces.

The transformation rules for this are straightforward. Each instance of Component is transformed to a corresponding extensional statement requiring a *Component* fact and *providesInterface* and *requiresInterface* facts for each element in *this.provides* and *this.requires*, respectively:

$$\begin{aligned} T^{UML}(Component) :=& (\{Component(this), providesInterface(this.provides), \\ & requiresInterface(this.requires)\}, \emptyset) \\ T^{UML}(Interface) :=& (\{Interface(this)\}, \emptyset) \end{aligned}$$

As mentioned above, we can construct Φ^M for a model M written in L by traversing the model and applying the transformation rules:

$$\Phi^M_{ext} = \bigcup_{m \in M} \bigcup_{mt \in metatype(m)} (replace(T^L(mt), m))_1$$
$$\Phi^M_{int} = \bigcup_{m \in M} \bigcup_{mt \in metatype(m)} (replace(T^L(mt), m))_2$$

whereas

- m is a model element in M
- $metatype(m)$ denotes the set of meta classes, that m is instance of.
- $replace(X, m)$ replaces in the set $X \subseteq EXT^{\tau_L} \times INT^{\tau_L}$ the variables m and $m.property$ as mentioned above.

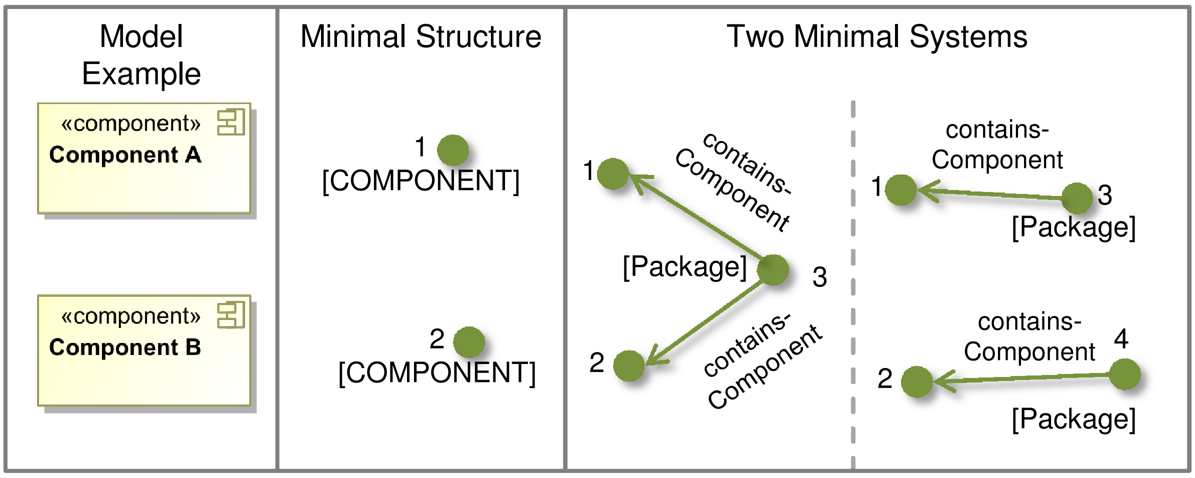


Figure 5.29: Minimal structures vs. minimal systems.

Hence, in the example model of Fig. 5.28, for the component C the expressions $Component(e_C)$, $providesInterface(e_C, e_{I_1})$, and $requiresInterface(e_C, e_{I_2})$ would be generated, as well as $Interface(e_{I_1})$ and $Interface(e_{I_2})$.

$metatype(m)$ contains not only the direct meta type of m, like Component for a component in a UML model but also the classes that are super classes of that direct meta type, which are specified by linking classes by the superClass association defined in M_3. Thus, if there were a transformation definition for the meta class Type, the statements defined in $replace(T^{UML}(Type), C)$ would be created, too. Analogously, the statements defined in $replace(T^{UML}(t), C)$ for every superclass t of Component would be generated.

5.5.2. Generation of Minimal Systems for Models

Compliance of models, as shown in Sec. 5.4, depends on the relationship between the sets of satisfying structures of both models' logical statements. It has been outlined that for the model, whose compliance with another model should be decided, the minimal satisfying structures are of special interest — if these can be extended to satisfying structures of the model to which compliance should be ensured, compliance between both models actually exists.

In general, it is not possible to do the check for compliance by model checking because the set of minimal systems that satisfies the statements of model is infinite. For example, the statement "There has to be at least one component" ($\exists x : Component(x)$) has infinite many, minimal satisfying structures. Furthermore, even if the minimal *structure* is unique, the set of minimal *systems* can still be infinite, as Fig. 5.29 shows. A model, stating that two certain components are required, has a unique minimal satisfying structure. The set of minimal systems is infinite; the minimal structure does not satisfy the axioms in Φ_{CBSD} but it can be extended to "larger" structures that do so and which are minimal systems.

Hence, model checking cannot be used efficiently in the general case covering all theoretically possible combinations of architectural, design, and implementation models. However, we can restrict generality to practically relevant cases by making certain assumptions about model properties:

1. Extensional statements of models are *unambiguously factual*.
2. Models are *intensionally complete*.
3. Extensional statements of models are *complete regarding* Φ_{CBSD} .

If these assumptions are true, there is a unique, minimal system satisfying the logical statements of a model. It can furthermore be derived from the extensional statements of a model.

Assumption 1: Extensional statements of models are unambiguously factual

The set Φ^M_{ext} contains all the extensional statements of a model M. Those statements can have different forms. For example, tautologies are extensional statements because they are true in every structure. Tautologies are also true in empty structure, hence as statement defined by a model they have no informative value regarding the systems that conform to the model. Every statement that does not use variables but only refers to constant symbols as terms for the arguments of predicates is extensional. The most relevant case for extensional statements defined by models is that those statements simply list facts described by the model like "Entity e_i is a component, and provides the interface represented by entity e_j". Modelled as a predicate logic formula, this would be a conjunction of facts.

More formally, this case could be described as a constraint over the elements of Φ^M_{ext}. For every $\varphi \in \Phi^M_{ext}$ holds that

$$\varphi := \bigwedge_{j=0,1,\dots} R_j(e_{i_1}, \dots, e_{i_n})$$

whereas every R_j is an n-ary relation symbol.

A model, for which every extensional statement has this shape, and for which Φ^M_{ext} is satisfiable at all, is called *unambiguously factual*. It describes only facts because every constraint that a statement puts on satisfying structures refers directly to a defined element of the universe (given by e_i's as predicate arguments). Furthermore, it does not describe unambiguous information like "e_i is either a component or an interface".

If every formula in Φ^M_{ext} is unambiguously factual, then the minimal satisfying structure for Φ^M_{ext} is unique. The universe of that structure consists of exactly those entities that are referred to by the constant symbols[10]. The relations consist of exactly those tuples that are described in the extensional formulae.

It is obvious, that this minimal satisfying structure can be easily generated from Φ^M_{ext}. The statements are traversed. In every statement, the operands of the conjunction are traversed. For every argument e_i of the predicate, an element in the universe is created, unless e_i has already been visited in one of the previous statements or operands. After that, a tuple is created for the

[10]see the definition of e_i in Sec. 5.3.

relation, interpreting the relation symbol given by the predicate name. The tuple consists of the elements created for the constant symbols.

In the case that M is an extensional model, this means that the minimal satisfying structure for M can be generated automatically.

Assumption 2: Models are intensionally complete

The first assumption alone is not enough to reduce the complexity in the case that an intensional model is subject to checking compliance with another model. Although, the minimal structure for Φ^M_{ext} is unique, the set of minimal structures for Φ^M can still be infinite. For instance, consider the observer pattern example from Sec. 5.4.2. If the, for example, the method bodies are not completely modelled, there might be many minimal satisfying structures for Φ^M. Nevertheless, the minimal satisfying structure is unique, given that Assumption 1 is true.

Hence, we assume in addition that the extension of a model is complete with regard to its own intensional constraints. This means, that the minimal satisfying structure for Φ^M_{ext} does also satisfy Φ^M_{int}. Whether this is true, can be ensured by model checking.

If the model is intensionally complete, the minimal satisfying structure for the extensional statements is also a satisfying structure for the whole model, because $s \models \Phi^M_{ext}$ and $s \models \Phi^M_{int}$ implies $s \models \Phi^M$. The structure s is also minimal regarding Φ^M. If there would be another minimal structure s' with $s' \models \Phi^M$ and s is an extension of s' then also $s' \models \Phi^M_{ext}$. As consequence, s could not be minimal regarding Φ^M_{ext}, leading to a contradiction. For the same reasons, s is also a minimal satisfying structure for Φ^M.

Thus, if the second assumption is true, we are able to generate the unique minimal satisfying structure for intensional models, too. The procedure is the same as described in the previous section.

Assumption 3: Models are conform to Φ_{CBSD}

As depicted in Fig. 5.29, the minimal satisfying structure is not necessarily a valid component-based system. It can violate axioms from Φ_{CBSD}. But this does not necessarily need to be the case. The statement “C is a component inheriting from interface I” is extensional and satisfiable by a minimal structure — but the same structure cannot be extended to a system according to Φ_{CBSD}, since inheritance between components and interfaces is not defined. In this case, the value of the minimal structure for the further compliance checking process is questionable because no valid system can be derived from it. On the other side, the infinite set of minimal systems cannot be used as input for model checking as mechanism to check compliance.

Hence we have to conclude, that the conformity of the extensional statements to Φ_{CBSD} has to be assumed. This means that the minimal satisfying structure for Φ^M_{ext} is also a satisfying system for Φ^M_{ext}.

It can be concluded that, given that all three assumptions hold, a unique, minimal satisfying system can be efficiently generated from the extensional statements that a model defines. The three assumptions restrict the set of models that can actually be checked for compliance. The models in that set are characterized by the fact, that they describe component-based systems according to Φ_{CBSD} (Assumption 3), that they are conform to their own, inherent constraints (Assumptions 2), and that the facts they describe are unambiguous (Assumption 1).

These extensional statements are generated from the model as described above. As a next step, the actual compliance checks can be performed.

5.5.3. Execution of Compliance Checks

A compliance check verifies whether a model M is compliant with another model M' according to Definition 18 (see Sec. 5.4.2). If M is compliant to M', then there is, following the definition, for every minimal system s of M a minimal system s' of M' such that we can build the union of s and s' and get a satisfying system for M.

If the three assumptions described in the previous section are true, s and s' are unique, and can be automatically generated. In addition to this generation, the union of the structures have to be generated, before model checking can be applied to determine whether the overall structure is a model of M'.

The procedure can be described in pseudo-code as illustrated in Listing 6.1. Input of the algorithm are two models M and M', and the sets of transformation rules that are applied either to M (T^L) or M' ($T^{L'}$) whereas L is the meta model of M, and L' is the meta model of M'.

First, the logical statements for both models are created (lines 11–12). If one of the sets contains formulas that are not uniquely factual, the procedure throws an error (lines 15–17). The unique minimal structure for the extensional statements of M is generated afterwards (line 18), and is checked (a) whether it is a component-based system and (b) whether it is intensionally complete (line 21–22). If one check fails, an error is thrown. After that, the minimal structure s' for M' is created and checked analogously (lines 23–27).

Finally, the structure u as union of s and s' is constructed (line 29). The result of the union operator is also folded. Normally, models have their own set of identifiers, thus, two models referring to the same logical entity attach two different identifiers to it. This entity would hence appear twice in the union of the models, or more precisely, in the union of their structures. For this reason, we apply a folding mechanism which defines for every unary relation symbol in the signature, when two elements of corresponding relations are considered to be identical — for example, two named elements are considered to be equal if they have the same qualified name.

Following the definition of compliance, the algorithm returns true, if and only if the merged structure satisfies Φ^M.

5.6. Summary

This chapter has explained the main building blocks of the formal framework for architecture compliance checking in component-based systems. To provide such a framework, there has to be a common formal representation of component-based systems and their models. This formal representation is based upon the common definitions of predicate logic. Component-based systems are structures conforming to a common signature τ_{CBSD} and obey the axiom system Φ_{CBSD}. The relation symbols in τ_{CBSD} reflect elements of component-based systems, like types, static structures of types, and behaviour specifications. Models of component-based systems

Listing 5.1: Algorithm to check architectural compliance

```
boolean checkCompliance($M$,$M'$,$T^L$,$T^{L'}$)
/* checks whether model M is compliant with model M' */
/* M is an instance of the meta model L */
/* M' is an instance of the meta model L' */
/* $T^L$ and $T^{L'}$ are the sets of transformation rules */
/* that are applied to $M$ and $M'$, respectively. */
/* Return 'true' if this is the case, 'false' if not. */
{
  /* Generate logical statements for both models */
  /* according to their transformation rules */
  $\Phi^M$ := generateStatements(M, $T^L$);
  $\Phi^{M'}$ := generateStatements(M', $T^{L'}$);
  /* Return error if statements for M or
     M' are not uniquely factual */
  if  (not isUniquelyFactual($\Phi^M$) ||
      not isUniquelyFactual($\Phi^{M'}$))
      return error
  Structure s := generateMinimalStructure($\Phi^M_{ext}$);
  /* Return error if M is not a system or
     not intensionally complete */
  if ($s \not\models \Phi_{CBSD}$ || $s \not\models \Phi^M_{int}$)
     return error;
  Structure s' := generateMinimalStructure($\Phi^{M'}$);
  /* Return error if M is not a system or
     not intensionally complete */
  if ($s' \not\models \Phi_{CBSD}$ || $s' \not\models \Phi^{M'}_{int}$)
     return error;
  /*Construct union of structures*/
  Structure u := fold(union(s,s'));
  if $u \models \Phi^M$ return true
  else return false;
}
```

are understood as logical statements over such structures; systems that satisfy the statement formulated by a model conform to the model.

The second building block is a classification of models based upon their level of abstraction. Three levels are distinguished. Architectural models express constraints upon systems that can be violated if the system is extended. Examples for such statements are architectural patterns. Design models define constraints that, if they hold in a system, cannot be violated by extending the system. However, removing elements that are not explicitly defined in the model, can lead to violated constraints. Implementation models, finally, are models formulating statements that stay valid in system no matter how it is changed.

Based upon this distinction, the term of compliance has been defined. For each model, there is the set of satisfying systems, τ_{CBSD}-structures for which the Φ_{CBSD}-axioms hold, and which satisfy the statements defined by the model under consideration. A subset is the set of systems that are minimal, i.e. systems from which no element can be removed without losing the property of satisfying the statements defined by the model. Compliance of a model M with a model M' is defined as the property that for each minimal model of M there is a minimal model of M' that way that both can be merged and result in another model of M'. Intuitively spoken compliance expresses that integrating the information that is contained in a refining model M into a more abstract model M' leads to a model that in which the intensional constraints of M' still hold.

As a third building block, the operationalization of compliance checks under the given definition of compliance is introduced. Since the concepts base upon checking whether a structure is a model (in the logical sense) for a logical formula, it seems natural to aim at applying model checking techniques.

The first step to operationalize compliance checks is the transformation of models into logical statements. The transformation rules are meta model-dependent. Hence, to define a language-independent procedure, we have to fix the language of meta models, i. e. the meta meta model. Assuming, that languages are meta models in the EMOF system, a procedure to apply transformation rules while traversing a model as instance of a given meta model, is introduced

In the second step, the minimal systems for the generated formulae are generated. To be able to do so, some assumptions about models with practical relevance have to be made; models have to be uniquely factual; models have to be intensionally complete; models conform to Φ_{CBSD}. Without these assumptions the set of minimal systems for a given model is in general infinite; if the assumptions apply, the minimal system for a model is unique. For the subset of models, for which the assumptions are true, an algorithm is introduced which checks two input models for compliance.

Chapter 6.

Architectural Rules for UML as Architecture Description Language

Contents

All space and matter, organic or inorganic, has some degree of life in it, and matter/space is more alive or less alive according to its structure and arrangement.

Christopher Alexander

The previous chapter has described the proposed formal framework for architectural compliance checking. To apply the approach to the case study (see. Chapter 4), several issues have to be addressed:

- The architectural rules for the architectural aspects of CoCoME described in Sec.4.2 have to be defined formally as logical formulae.

- An Architecture Description Language (ADL) has to be selected to model the architectural aspects that were only described textually and informally in [RRPM08]. Its transformation into logical expressions, including into the architectural rules mentioned above, has to be defined.

- A transformation of UML models into logical expressions has to be defined.

If solutions are provided for these issues, the developed reference design model can be checked for compliance with the architectural model written in the selected ADL.

For this task, UML was selected as ADL. Since version 2.0 of the specification, UML provides some concepts to model architectures, like components, connectors, etc. Furthermore, there exist many approaches to further enhance UML's capabilities to model software architectures (e.g. [RKJ04, LCM06, MKB$^+$08]) as well as work which shows how renowned and dedicated ADL like ACME, COSA, or others can be mapped to UML [GeA03, KSO06, Oqu06].

To model advanced architectural concepts which imply architectural rules, UML profiles are used. UML profiles provide a light-weight mechanism to extend the UML meta model and hence to introduce new language elements. It allows the user of UML to tailor the language according to its own specific needs. Profiles define so-called *stereotypes* to extend existing language elements and to introduce specific terminology or notations, which are used instead of the predefined, standard language elements.

Profiling is used in a couple of approaches to integrate architectural concepts into UML, e.g. architectural primitives or patterns [ZA08, RKJ04, MKB$^+$08, KA08]. We will use profiles to define concepts required to model the architectural aspects of CoCoME, as described in Sec. 4.1.2. Architectural rules are attached to the stereotypes contained in the profile.

However, it is important to note that the specific ADL, or how UML is exactly extended for the purpose of architecture description, is not of special interest here. The architectural rules are specified in terms of τ_{CBSD}, and can hence be used in the transformation definition according to Sec. 5.5.1 for any ADL.

The following subsections are structured as follows. Section 6.1 explains the main principles of the UML profile mechanism.

In Sec. 6.2 the architectural rules for the three main architecture aspects of CoCoME will be introduced: layers, interfaces with copy-semantics, and event-based architecture. For each single architectural aspects, a set of stereotypes will be defined that are required to model it at the architectural level or to refine it in a design model. For each stereotype, the rules, either architectural or design, will be formally developed as logical expressions.

In Sec. 6.3, the transformation definition to generate logical statements for UML models will be presented. One focus is the transformation of UML model elements used in class diagrams, component diagrams, and composite structure diagrams to model the static structure of systems. The second focus is the transformation of sequence diagrams and the elements used there to specify the behaviour of systems. Section 6.4 concludes the chapter.

6.1. UML Profiling Mechanism

As a general purpose language, UML tries to cover as many aspects of software system development as possible, and to be as generic as possible. It is quite natural, that there are often situation in projects when certain specific issues cannot be modelled in an intuitive way or cannot be expressed at all. Furthermore, innovations require often a certain time to find their way into the UML standard. As a consequence, UML, just like any modelling language, will probably always be incomplete to a certain degree.

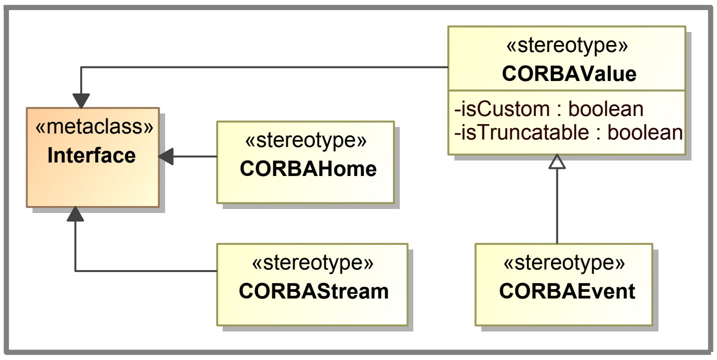


Figure 6.1: UML Profile Example (from [Obj08]).

There are two ways to tailor UML to project- or user-specific needs. First, the meta model of UML can be modified, i.e. the language definition is changed. This is also known as *heavyweight extension* of UML. While this is a rather powerful and flexible way to extend UML, because it is possible to introduce new elements, delete old ones or change their properties, it has also strong practical implications. Changes have to be communicated; developers familiar with standard UML have to learn the changes to the language and get used to it; the used UML modelling tools have to support the modification of the standard, what is most often not the case, or have to be adapted, what is often not possible or time-consuming and expensive. Hence, the flexibility, that comes with the modifiability of the meta model, does not pay off in many cases.

As a second option, the so-called profiling mechanism of UML can be used. A profile defines project- or domain-specific extensions of the UML meta model that add certain elements to the meta model or restrict the way how language elements may be used in the project or domain. Profiles are models at the MOF level M1, and can be dynamically applied to models, indicating that the restrictions of the domain or project have to hold in the models. The advantage is that the meta model of UML is not changed. Furthermore, most of the sophisticated UML modelling tools support the creation and application of profiles.

Formally spoken, a profile is a special variant of a UML package. Figure 6.1 shows an example from the CORBA profile defined by the OMG [Obj08]. A profile defines so-called stereotypes. Stereotypes describe how existing meta classes can be extended, and allow using platform or domain specific concepts, terminology, or notations, in places where the extended meta classes could be used. For instance, Fig. 6.1 shows the CORBAHome stereotype, indicating a special kind of interfaces, the home interfaces, which provide standard functionality to components in CORBA, like instantiating, finding instances, etc [BW01]. The extension relation is modelled by an arrow pointing to the extended meta class. In UML models, where a stereotype is applied to an instance of a meta class, like an interface, the stereotype is denoted in guillemets above the element's name.

A stereotype can define attributes to add domain specific information to the meta class it extends. For example, the boolean attribute of CORBAValue in Fig. 6.1 determines whether the stereotyped interface represents a value type in CORBA whose values can be truncated (see [BW01] for details).

Another important part of profiles are the constraints attached to stereotypes and, hence, to the domain specific extensions that are made by the profile. They describe boundary conditions that have to hold if a model defines an element having the stereotype. For example, the specification of the CORBA profile states that a home interface (remember, an interface stereotyped with CORBAHome) can only inherit from at most one home interface. This is specified formally in [Obj08] as OCL constraint, and can be checked on models applying the profile by adequate UML tools.

Stereotypes can also be specialized by, and only by, stereotypes. The meaning is basically the same as specialization between classifiers in UML; in places where the more general stereotype can be applied, also its specializations can be applied.

In the following, UML profiles will be used to introduce aspects of architecture and design that specify architectural or design rules for the development of a system. There will be two profiles, the *Architecture Profile* and *Design Profile*, one for each single class of rules.

6.2. Definition of Architectural Rules for CoCoME

Three different architectural aspects needs to be modelled in an architectural model of the CoCoME case study, and each defines a corresponding set of architectural rules. First, modelling of layers will be introduced and the architectural rules that such a structure defines, as explained in see Sec. 4.2.1, will be formalized. In the case study, these rules apply to the general structure of the Inventory System. Second, the elements to model components/interfaces with copy semantics are introduced, and their rules are formally defined (see 4.2.2). These rules apply to the components of the application layer in the case study. Last but not least, the event-based architecture (see Sec. 4.2.3) and its rules will be considered. In the case study, the cash desk system must be compliant with those rules.

6.2.1. Layers

Section 4.2.1 informally described the rules that a layered architecture defines for the refinement of the system. In conclusion, those rules are:

- Layers may only depend on other layers, if these are lower in the hierarchy, and.
- if there is an “allowed to use” relationship between the specific layers.
- Static dependencies arise from provide/require relationships between components and interfaces, from usage or inheritance between interfaces, or from typing of parts between components.
- Dynamic dependencies arise from method calls between classifiers whereas, due to polymorphism, the dependency is not visible at design-time in general.

Figure 6.2 shows the fragment of the architecture profile containing the stereotypes required to model layers. The Layer stereotype is modelled as specialization of the stereotype Group which extends the UML meta class Package.

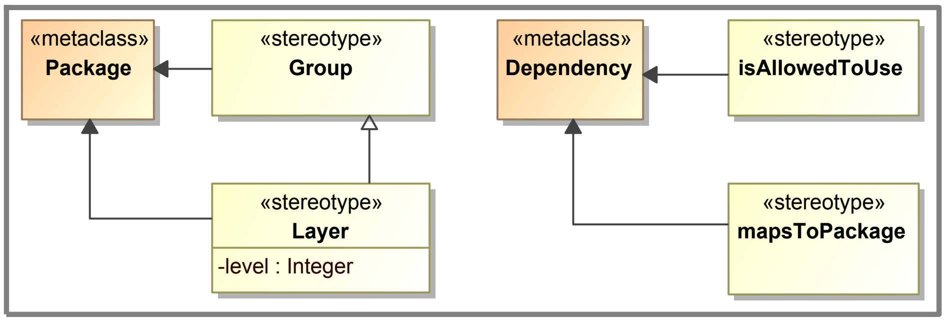


Figure 6.2: UML Architecture Profile: Stereotypes to model grouping of elements (layers).

As formulated in [ZA08], components are often composed in a set which does not represent a component itself; it does not manifest itself as instance at runtime but is only the sum of its parts. As such, these sets are virtual and called groups, simply providing a namespace for model elements.

Packages are the common language elements in UML to provide namespace, and can contain “packageable elements” like classifiers (see [Obj10b]). As such, it is the prime candidate to be the base class for the Group stereotype.

The profile defines the additional constraint for groups that they do not own elements to express that they are virtual, but they only import elements. This means that components, interfaces, etc., are *defined* in a normal package and imported into the layer. The actually imported packages are defined by mapsToPackage dependencies (see below).

The Layer stereotype adds the level attribute to packages. It indicates the position of the package in the layer hierarchy. Additional constraints ensure that dependencies directed from a layer package are only allowed if the target of the dependency, in case it is stereotyped as layer, has a lower level than the source.

It makes sense not only to introduce the Layer stereotype alone but also the more general stereotype Group. Models of layers of a software system do not only contain the layers in many cases, but also other groups of components, like segments [CBB+10]. These can be modelled using the Group stereotype or, if necessary, by specializing it.

The stereotype mapsToPackage extends the meta class Dependency of the UML meta model. A dependency denotes a directed relationship in UML, modelling that the element at its source uses the element at the target for its specification or implementation (see [Obj10b]). A dependency stereotyped with mapsToPackage has always a group as source and a package as target, whereas the target package is stereotyped neither with Group nor Layer. The dependency indicates that all elements contained in the target package are virtually grouped in the group at the dependency source. Figure 4.8 (see p. 49) depicts an example of how to this mapping of layers can be applied.

The last stereotype to model layers is isAllowedToUse which models allowed-to-use relationships between groups. A dependency stereotyped this way always connects groups, i.e. packages stereotyped with Group or Layer. Notice, that the rule that dependencies starting at a layer must target at a lower layer also holds for dependencies stereotyped with isAllowedToUse, hence it is syntactically not correct to use the stereotype in other cases.

To represent layers and related concepts also formally in component-based systems, we define a signature $\tau_{arch} \supseteq \tau_{CBBSD}$ which contains all relation symbols that can be used in statements representing UML models that apply the architecture profile.

τ_{arch} contains the following relation symbols:

- $Group \subseteq Package$
- $Layer \subseteq Group$
- $mapsToPackage \subseteq Layer \times Package$
- $isAllowedToUse \subseteq Group \times Group$

whereas the axioms regarding the domain of relation symbols apply (see Sec. 5.3.1).

The transformation of model elements stereotyped with one of the elements of the architecture profile into an extensional statement is straightforward; a stereotyped package p is transformed into $Layer(id_p)$ or $Group(id_p)$, depending on the actual stereotype. Notice, that the relation symbol domain axioms ensure that in these cases also $Package(id_p)$ has to be true. Stereotyped dependencies are transformed into $mapsToPackage(id_s, id_t)$ and $isAllowedToUse(id_s, id_t)$, respectively, depending on the stereotype. id_s represents the entity at the source, id_t the entity at the target of the dependency, respectively.

These symbols are used to define the architectural rules that layers define. Notice, that the first rule, dependency from top to bottom, is syntactically ensured. Since we assume that syntactically correct models only are checked for compliance, we do not formulate a logical expression for that part. In the following, the rule that is checked for each single layer will be developed top-down.

The architectural rule for a group (or layer) l is defined as follows:

$$T^{UML,arch}(Group) := \{\ldots, \neg\, illegalDependenciesFromGroup(this)\} \tag{6.1}$$

whereas *illegalDependenciesFromGroup* is true, if there exist dependencies that do not conform to an existing *isAllowedToUse* tuple:

$$\begin{aligned} illegalDependenciesFromGroup(g) :=& \exists g' : (groupDependency(g, g') \wedge \\ & \neg\, isAllowedToUse(g, g') \wedge \neg\,(g' = g)) \end{aligned} \tag{6.2}$$

A group dependency between g and g' exists if g contains an element which depends on an element contained in g'.

$$groupDependency(g, g') := \exists e \exists e' : (inGroup(e, g) \wedge inGroup(e', g') \wedge dependsOn(e, e')) \tag{6.3}$$

whereas *inGroup* reflects the containment relationship. An element is contained in a group, if it is directly contained in a package which the group maps to, or one of the package's subpackages. This also means that elements of groups are either components or interfaces, the only elements that are contained in packages by the definitions of τ_{CBSD}:

$$\begin{aligned} inGroup(e, g) :=& \exists p : (mapsToPackage(g, p) \wedge (containsComponent^{+}(p, e) \vee \\ & containsInterface^{+}(p, e))) \end{aligned} \tag{6.4}$$

Now, it has to be clarified and expressed, which relationships actually constitute a dependency between components and interfaces. In expression (6.3), this is abstracted by *dependsOn*(e, e'). This is defined as

$$dependsOn(e, e') := \bigvee_{i=1,\dots,8} dependsOn_i(e, e') \tag{6.5}$$

Each $dependsOn_i$ defines one way elements can depend on each other. The expression $dependsOn_1(e, f)$ assumes that e is a component, and f is an interface. It expresses that e depends on f if it provides the interface or requires it. The reason is that the specification of e uses the specification of f. Hence the definition is as follows:

$$dependsOn_1(e, f) := providesInterface(e, f) \lor requiresInterface(e, f) \tag{6.6}$$

The formula corresponds to the *usage between components and interfaces* as mentioned in Sec. 4.2.1. As second issue and dependency reasons *specialization between types* was mentioned. Hence, extensions between interfaces have to be considered:

$$dependsOn_2(e, f) := Interface(e) \land Interface(f) \land extends^+(e, f) \tag{6.7}$$

Furthermore, in Sec. 4.2.1 the *usage of types* in different forms has been identified as source for dependencies. One form is the usage of an interface as type of a member variable:

$$\begin{aligned} &dependsOn_3(e, f) := Interface(e) \land Interface(f) \land \\ &\quad \exists m : (hasMember(e, m) \land hasType(m, f)) \end{aligned} \tag{6.8}$$

Interfaces can define signatures that use another interface as type for one of its parameters which constitutes another form of usage of types. This case is expressed by

$$\begin{aligned} &dependsOn_4(e, f) := Interface(e) \land Interface(f) \land \\ &\quad \exists s \exists p : (hasSignature(e, s) \land hasParameter(s, p) \land hasType(p, f)) \end{aligned} \tag{6.9}$$

In case e and f are components, it is possible that e encapsulates a part typed with f. This is also a dependency:

$$\begin{aligned} &dependsOn_5(e, f) := Component(e) \land Component(f) \land \\ &\quad \exists p : (encapsulates(e, p) \land hasType(p, f)) \end{aligned} \tag{6.10}$$

The last case of dependencies mentioned in Sec. 4.2.1 is the method call relation between components. It is reflected by the following two formulae. A component A can also depend on a second component B if there exist two parts, one typed by A, one typed by B, and a connector connecting them which is "used synchronously". This means that, assuming the A-typed part is the source, that the implementation of A calls methods synchronously on the required port being source of the connector. The same situation occurs if the A-typed part is inside a component C, and a request is delegated to a required port of C which is connected to B. Figure 6.3 illustrates both cases. The first case is expressed formally by formula (6.11); the second case is formalized by formula (6.12).

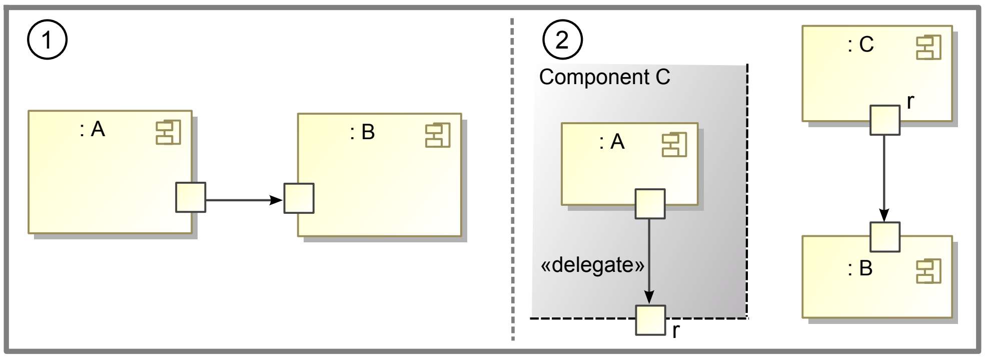


Figure 6.3: Situations, in which a dependency between components A and B exists, due to synchronous message calls via connectors.

$$\begin{aligned}
&dependsOn_6(e,f) := Component(e) \wedge Component(f) \wedge \\
&\quad \exists con \exists p \exists q \exists r : (InnerPart(p) \wedge InnerPart(q) \wedge Connector(con) \wedge hasType(p,e) \wedge \\
&\quad (hasType(q,f) \vee \exists t : (Class(t) \wedge containsClass(f,t) \wedge hasType(q,t))) \wedge \\
&\quad sourceContext(con,p) \wedge targetContext(con,q) \wedge connectorSource(con,r) \wedge \\
&\quad \exists c \exists m \exists s : (containsClass(e,c) \wedge hasMethod(c,m) \wedge \\
&\quad\ containsStatement(m,s) \wedge SynchronousInvocation(s) \wedge invokedAt(s,r)))
\end{aligned} \tag{6.11}$$

The first four lines describe the typing of required elements and their relationships to the considered connector. *con* is a connector between two contextual parts p (source) and q (target), p is typed with the component e; q is either typed with f of with a class contained in f. The source port of the connector is r. The last two lines express, that there has to be class in e, the potentially dependent component, which calls a method m at the required port r.

Notice, that this definition also covers the variant of the first case in which the target context of the connector is a component-local class; in this case, there is a dependency between the type of the source part, and the surrounding component (i.e. C in situation 1, Fig. 6.3).

The following expression (6.12) is composed as follows. The first four lines express, that *del* is a delegation connector, connecting the (required) ports rp as source and r as target. Source context is a part typed with the component e. Lines 5–6 express that a connector has to be present with source port r; its target context is either a part typed by f directly, or typed by a class contained in f. Lines 7–8 express the required synchronous call to rp to constitute a dependency.

$$\begin{aligned}
&\mathit{dependsOn}_7(e,f) := \mathit{Component}(e) \wedge \mathit{Component}(f) \wedge \\
&\quad \exists del \exists rp \exists p \exists q \exists r : (\mathit{Connector}(del) \wedge \mathit{isDelegator}(del, \text{“true”}) \\
&\quad \mathit{sourceContext}(del, p) \wedge \mathit{hasType}(p, e) \wedge \mathit{connectorSource}(del, rp) \\
&\quad \mathit{connectorTarget}(del, r) \wedge \exists con : \\
&\quad (\mathit{connectorSource}(con, r) \wedge \mathit{targetContext}(con, q) \wedge \\
&\quad (\mathit{hasType}(q, f) \vee \exists t : (\mathit{Class}(t) \wedge \mathit{containsClass}(f, t) \wedge \mathit{hasType}(q, t))) \wedge \\
&\quad \exists c \exists m \exists s : (\mathit{containsClass}(e, c) \wedge \mathit{hasMethod}(c, m) \wedge \\
&\quad\; \mathit{containsStatement}(m, s) \wedge \mathit{SynchronousInvocation}(s) \wedge \mathit{invokedAt}(s, rp))))
\end{aligned} \tag{6.12}$$

Furthermore, an object can call methods on references, typed by interface, that were returned before by a method call. Knowing only their interface by specification, the calling object can depend on implementations dynamically, without a static dependency in the specification.

This is concluded in the following rule: a component e depends upon a component f if e contains a class c which specifies a synchronous call on a reference variable which is possibly dynamically typed with a class d contained in f. This is stated by the following rule:

$$\begin{aligned}
&\mathit{dependsOn}_8(e,f) := \mathit{Component}(e) \wedge \mathit{Component}(f) \wedge \\
&\quad \exists c \exists d \exists m \exists r \exists s : (\mathit{containsClass}(e, c) \wedge \mathit{hasMethod}(c, m) \wedge \\
&\quad \mathit{SynchronousInvocation}(s) \wedge \mathit{containsStatement}(m, s) \wedge \mathit{invokedAt}(s, r) \wedge \\
&\quad\; \mathit{dynamicType}(r, d) \wedge \mathit{containsClass}(f, d))
\end{aligned} \tag{6.13}$$

The definition of the architectural rule for the stereotype Group is sufficient to describe the architectural aspect of layering — the other stereotypes introduced so far do not specify additional rules.

We only show an intuitive proof that the statement for a particular layer/group is indeed intensional and non-local, i.e. it is an architectural rule. In expression (6.2), it is stated what constitutes an illegal dependency — there exists another group with a certain property (used but not allowed to be used). This property mainly builds upon the dependency properties of elements that are contained in the group defined in the following expressions. Hence, just because a group consists of elements that do not illegally depend on anything, it is not sure that this could not be changed by adding new elements to the group. Thus, the rule cannot be local (see Sec. 5.2) and must be non-local.

Sections 7.1 and 7.3.1 describe to which specific architectural rules the layered architecture of the CoCoME case study leads, and how they are modelled during architecture and design development.

Remarks on Static Analysis

The definition of *dependsOn*$_8$ (see (6.13)) shows that a full analysis of dependencies between layers or groups requires knowledge about dynamic properties of the system; due to polymorphism, the actual type of an object at runtime can be different from the declared type. This

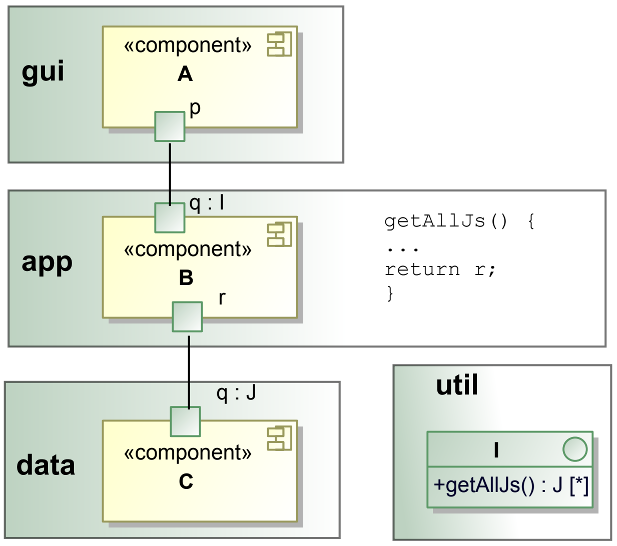


Figure 6.4: Unwanted dependency between GUI layer and data caused by dynamic type dependency.

can lead to unwanted dependencies in a layered architecture. Consider the layering depicted in Fig. 6.4. The interface J, defined in the lowest layer util, is realized by the components B and C. This conforms to the architecture since the containing layers are allowed to access util. Component B provides an interface used by A. This interface provides a method getAllJs() which returns a set of instances of J. So far, everything is fine and compliant with the layering.

But what, if the method body contained in a classifier of B that realizes getAllJs() retrieves the instances it returns from component C by calling getSomeJs():J on its required port? It is possible that the method retrieves instances defined in C and passes it to A. This leads to a dependency between layer gui and data which is not architecturally compliant.

For that reason, *dependsOn*$_8$ contains the predicate *dynamicType*, which refers to the dynamic type of an object. However, the determination of the dynamic type of a variable (as a representative of an object) at design time results always in an approximation of the actual dynamic type at runtime. Due to polymorphism the actual dynamic type of a variable cannot be known at design time in general, since it depends in general on the actual execution path on which a variable is initialized and target of assignment statements, like reference passing or returns from method calls (see also Sec. 5.3.5). Since a complete investigation cannot be done efficiently, the possible dynamic types have to be approximated — this means that all algorithms for static type analysis can return too many results in certain situations. In general, there is a trade-off between the accuracy and the scalability of type analysis algorithms.

Well-known approaches are the *Class Hierarchy Analysis (CHA)* [DGC95] and *Rapid Type Analysis (RTA)* [Bac97]. CHA takes the inheritance hierarchy of a program into account to determine the type of a variable. Every subtype of the variable's declared type is assumed to be a possible dynamic type. RTA refines this and considers only instantiated subtypes as possible types. Important to notice is that both approaches do not consider inter-procedural relations in programs. This means that a program is basically a single method that is analyzed. The results of the analysis do not take the implementation of called methods into account. Those algorithms have low runtime and space complexity but are also very imprecise for complex systems.

More precise are algorithms that also consider the inter-procedural control flow in a system. These approaches keep track of caller and called methods by constructing call graphs. This structure is used to determine dynamic types more precisely. However, in object-oriented, and hence in component-based systems, it is difficult to construct a call graph due to polymorphism — compared to the procedural programs it adds to complexity to determine the target method of a call `x.foo()` if the dynamic type of `x` cannot be determined precisely. This means that call graph construction depends on type analysis and vice versa. As a result of this fact, most algorithms to inter-procedural type analysis require multiple analysis iterations stopping at a fix-point. This leads in general to a worse complexity but more precise results compared with the approaches described above. There are several approaches that try by different means to restrict the number of iterations [GC01, Ple96, SHR$^+$00, TP00].

Each of these algorithms can in principal be used to determine the elements in the relation $dynamicType$ used in formula (6.13). Hence, the definition of the predicate is intentionally omitted. The selected "implementation" may depend on context-specific weighting of performance and precision.

Another useful static analysis technique for architectural compliance checking is *pointer* or *points-to* analysis [Hin01]. Originally, the goal of points-to analysis is to decide which references can point to which storage location, often with the purpose to decide whether two variables can point to the same object. The most popular algorithms for points-to analysis are from Andersen [And94] and Steensgard [Ste96], a good overview of recent research is given in [Ram04].

The question whether two variables point to the same object arises in the definition of architectural rules for layers in expression (6.12). With the given definition, we are able to detect dependencies that exist due to method calls on required port, for example `rp.foo()"` whereas `rp` is a required port. The expression does not notice a dependency if this statement is replaced by `"v=rp; v.foo()"`, although `v` will point to the same object. Although this is a severe restriction in practice, we will omit the introduction of points-to analysis for now to keep focus on the definition of architectural rules.

6.2.2. Service-Oriented Layer Interface

According to Sec. 4.2.2, the following rules regarding the service-oriented interface of the application layer have to be ensured:

1. The access from the GUI layer to the application layer is limited to calls of methods defined in service components; these are components providing at least one service interface.

2. A service interface is an interface defining service methods only. These are methods using only transfer object classes and primitive types for parameters and return value types. Returned objects are always created during the call of the service method.

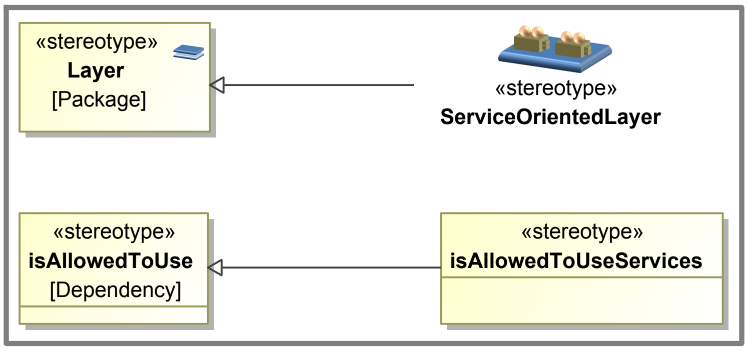


Figure 6.5: Stereotypes in the Architecture Profile to model service-oriented access to layers.

3. Transfer objects refer only to other transfer objects or data entities of primitive types.

In contrast to the representation of the layers aspect in the developed profiles, the service-oriented architecture aspect is represented in both, the architecture *and* the design profile. The reason for this is simple. As we will see, a layer with service-oriented interface implies an architectural rule about the elements contained in the layer and how they should be used. Hence, it must be possible to model such a layer in an architectural model. The elements to model services, transfer objects, etc. — the elements that actually make up a service-oriented layer, however, constitute design rules, i.e. intensional but local statements about valid systems.

Thus, there is a clear separation of design responsibilities between software architect and software designer; the software architect can model that a certain layer has to be accessed in a service-oriented manner; the designer models which services and transfer objects are actually necessary to provide the required functionality.

The architecture profile contains the stereotype ServiceOrientedLayer as depicted in Fig. 6.5. It specializes Layer and hence inherits all of Layer's constraints; furthermore, a package stereotyped with ServiceOrientedLayer is also transformed into architectural rules according to the transformation definition for Group (see Sec. 6.2.1). In addition, it will be transformed into a statement that formally states the first rule from the enumeration above. Furthermore, the stereotype isAllowedToUseServices is defined, specializing isAllowedToUse, that connects layers (or groups), whereas it indicates that the layer at the source is allowed to use only services of the layer at the target.

While an element stereotyped as service-oriented layer in an architecture model basically means that access from other layers is only permitted by using services, the design profile enables us to actually model services. The constraints that apply to the corresponding stereotypes describe how a service is actually defined. Figure 6.6 depicts the three relevant stereotypes and the extended UML meta classes. ServiceComponent marks components that provide at least one service interface. Service marks service interfaces containing only service methods. Transfer marks classes of transfer objects.

One might have noticed that the second and third rule of the rule enumeration above might be expressed as OCL constraints attached to the corresponding stereotypes. For example, it is easy to express that a service component must provide at least on interface with the property that it is stereotyped with Service. Thus, it seems that there is no need to express a

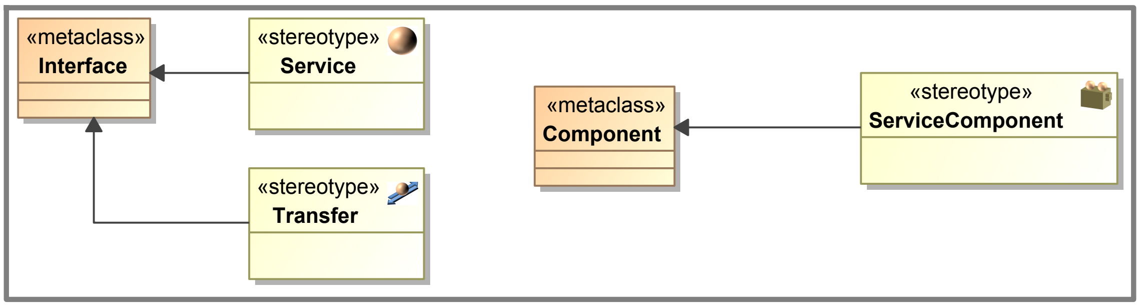


Figure 6.6: Stereotypes in the Design Profile to model transfer objects and services.

corresponding transformation definition for the stereotype. Regarding layers, we did the same with the constraint describing that accesses between layers are allowed only downwards the layer hierarchy. However, the difference is that layers are modelled in architectural models as separate packages only; hence a corresponding "model-local" OCL constraint can be checked. In case of the three stereotypes of Fig. 6.6, they refer to elements that will be referenced in refining implementation models, too; a service component modelled in the design model will somehow manifest in code. This code, an implementation model, has to conform to those constraints — which is exactly following the design rules given in the design model.

There is no implementation model in the scenarios, in which the overall compliance checking approach is demonstrated. Hence, the design rules are not checked to show that a separate implementation model conforms to them. Nevertheless, they need to be checked to show that the design model itself, being checked for compliance with an architecture model, is intensionally complete (see Sec. 5.5.2). In the following, the architectural rules for service-oriented layers will be developed bottom-up, starting with the design rules that service components, service interfaces, and transfer object interfaces define.

For UML models to which the design profile is applied, we specify the signature $\tau_{design} \supset \tau_{CBSD}$, which includes the following relations symbols:

- $Transfer \subseteq Interface$
- $Service \subseteq Interface$
- $ServiceComponent \subseteq Component$

The transformation of model elements stereotyped in one of those three ways is straightforward and maps one-to-one on a statement like $Stereotype(id_{me})$, e.g. if m is an interface stereotyped as service, the extensional statement $Service(e_m)$ will be generated for it.

For the intensional part of the transformation definition, we will take a look on the design rules that transfer objects have to follow. The rules are specified for interfaces marked with the stereotype Transfer. These interfaces define the members containing the data to be transferred. The transformation definition is as follows:

$$T^{UML,design}(Transfer) := (\{\ldots\}, \{onlyLegalMembers(this), onlyGettersAndSetters(this)\}) \tag{6.14}$$

The expression *onlyLegalMembers* ensures that members are typed by either primitive types or interfaces that model transfer objects. It is defined as follows:

$$\begin{aligned} \mathit{onlyLegalMembers}(x) :=& \forall y : (\mathit{hasMember}(x, y) \rightarrow \exists t : \\ & (\mathit{hasType}(y, t) \wedge (\mathit{PrimitiveType}(t) \vee \mathit{Transfer}(t)))) \end{aligned} \tag{6.15}$$

Note, that, although *onlyLegalMembers*(*this*) makes a universal quantified statement, the statement is local. The reason is that the only way to change a satisfying interpretation of the formula into an unsatisfying one by additions is to add an element to the relation for *hasMember* — by some *y* which is member of *this* but for which the right side of the implication is false. However, *hasMember* is in the set of close relations and; thus, it does not affect the locality of the statement.

onlyGettersAndSetters is a placeholder to define that only methods in transfer objects are allowed that set the value or members or get the value. Since τ_{CBSD} allows direct access to members, in contrast to most object-oriented programming languages in which direct access should be avoided, there is a simple way to disallow methods different than this — there are no methods allowed at all. We keep this short definition to avoid a lengthier and more general definition, which would specify the form of getters and setters more precisely.

Hence, we define for the sake of clarity and simplicity:

$$\mathit{onlyGettersAndSetters}(x) := \neg\exists y : \mathit{hasSignature}(x, y) \tag{6.16}$$

The transformation for elements stereotyped with Service is defined as follows:

$$T^{UML,design}(\mathit{Service}) := (\{\ldots\}, \{\mathit{hasServiceMethodsOnly}(\mathit{this})\}) \tag{6.17}$$

hasServiceMethodsOnly expresses that every signature of the interface is a service method:

$$\begin{aligned} \mathit{hasServiceMethodsOnly}(x) :=& \forall y : (\mathit{hasSignature}(x, y) \rightarrow \\ & \mathit{isServiceSignature}(y)) \end{aligned} \tag{6.18}$$

isServiceSignature states that the signature of service methods may only use types that are either primitive or stereotyped as Transfer:

$$\begin{aligned} \mathit{isServiceSignature}(x) :=& \forall y : (\mathit{hasParameter}(x, y) \rightarrow \\ & \exists t : (\mathit{hasType}(y, t) \wedge (\mathit{PrimitiveType}(t) \vee \mathit{Transfer}(t)))) \end{aligned} \tag{6.19}$$

Since interfaces do not yet provide an implementation for signatures, the constraint that every returned transfer object must be new, must be generated for the actual implementation of a service — to be more precise, for the service components and the contained classes. Hence, for service components the transformation is defined as follows:

$$T^{UML,design}(\mathit{ServiceComponent}) := (\{\ldots\}, \{\mathit{providesService}(\mathit{this}, \mathit{this.provided})\}) \tag{6.20}$$

providesService states that the component provides a port for the service interface specified in the second parameter. That port must be typed with a class implementing the service correctly regarding return values:

$$\begin{aligned} \mathit{providesService}(x,y) := {} & \mathit{ServiceComponent}(x) \land \mathit{Service}(y) \land \\ & \mathit{providesInterface}(x,y) \rightarrow \\ & \exists p \exists t : (\mathit{ProvidedPort}(p) \land \mathit{hasType}(p,t) \land \\ & \mathit{implements}^{+}(t,y) \land \mathit{classImplementsService}(t,y)) \end{aligned} \tag{6.21}$$

classImplementsService has to express the following: for every method of the class that has a signature defined in a service interface, one of the two possibilities to return transfer objects, in case an interface stereotyped with Transfer have to be realized:

$$\begin{aligned} & \mathit{classImplementsService}(x,y) := \mathit{Class}(x) \land \mathit{Service}(y) \land \\ & \quad \mathit{implements}^{+}(x,y) \rightarrow \\ & \quad (\forall m \forall s : \mathit{hasMethod}(x,m) \land \mathit{implementsSignature}(m,s) \land \\ & \quad \mathit{hasSignature}(y,s) \rightarrow \\ & \quad (\exists t : \mathit{hasType}(s,t) \land \mathit{Transfer}(t) \rightarrow \\ & \quad (\mathit{returnsCreatedTO}(m) \lor \mathit{returnsForwardedTO}(m)))) \end{aligned} \tag{6.22}$$

returnsCreatedTO describes the case that the method creates a transfer object and returns it. The method must contain an instance creation statement that passes the new transfer object to a local variable, which is returned:

$$\begin{aligned} & \mathit{returnsCreatedTO}(x) := \exists c \exists d \exists r \exists t \exists v : (\mathit{containsStatement}(x,c) \land \\ & \quad \mathit{InstanceCreation}(c) \land \mathit{containsStatement}(x,r) \land \mathit{Return}(r) \land \\ & \quad \mathit{instantiatedClassifier}(c,d) \land \mathit{implements}^{+}(d,t) \land \mathit{Transfer}(t) \land \\ & \quad \mathit{LocalVariable}(v) \land \mathit{assignedTo}(c,v) \land \mathit{returnedValue}(r,v) \land \\ & \quad \mathit{successorNode}^{+}(c,r)) \end{aligned} \tag{6.23}$$

returnsForwardedTO states that the result of a method call can be used as return value if the called method is implementing a signature from a service interface:

$$\begin{aligned} & \mathit{returnsForwardedTO}(x) := \exists c \exists i \exists r \exists s \exists t \exists v : (\mathit{containsStatement}(x,c) \land \\ & \quad \mathit{SynchronousInvocation}(c) \land \mathit{containsStatement}(x,r) \land \mathit{Return}(r) \land \\ & \quad \mathit{invokedSignature}(c,s) \land \mathit{definesSignature}(i,s) \land \mathit{Service}(i) \land \\ & \quad \mathit{LocalVariable}(v) \land \mathit{assignedTo}(c,v) \land \mathit{returnedTo}(r,v) \land \\ & \quad \mathit{successorNode}^{+}(c,r)) \end{aligned} \tag{6.24}$$

It is important to note that these rules do not match for all possible and correct implementations of transfer objects or services. Consider, for example, an implementation of a service which takes the returned objects from a pool of available objects. The service implementation ensures that an object from the pool is never used twice as return value. It can easily be seen that this would be an implementation conforming to the informal stated rule of Sec. 4.2.2 that a transfer object is for exclusive use of one client.

However, this shows the borders of expressiveness of τ_{CBSD} statements. Since τ_{CBSD} does not formalize runtime constructs like objects, thread, or execution paths, behavioural rules about such constructs are hard or impossible to express. Section 9.2 will discuss these issues in more detail as future work.

Concluding the previous statements, we have now realized the last two of three rules mentioned in the beginning of this subsection; it is defined what services are, and rules for the implementation, given by service component, are defined; it is specified, how transfer objects are structured. Hence, we can finally define the transformation of packages that are stereotyped by ServiceOrientedLayer and used in architectural models.

The signature τ_{arch} defines the relation symbols *ServiceOrientedLayer* $\subseteq$ *Layer* and *isAllowedToUseServices* $\subseteq$ *isAllowedToUse*. For models, to which the architecture profile is applied, the corresponding transformation definition is given:

$$\begin{aligned}
&T^{UML,arch}(isAllowedToUseServices) := \\
&\qquad (\{isAllowedToUseServices(this.source, this.target)\}, \emptyset) \\
&T^{UML,arch}(ServiceOrientedLayer) := \\
&\qquad (\{ServiceOrientedLayer(this)\}, \{onlyServicesExternallyUsed(this)\})
\end{aligned} \tag{6.25}$$

The architectural rule will specify that if a method of a component defined in a service-oriented layer is called, and if the source of the call comes from a different layer, then the target component must be a service component. Therefore, the rule is defined informally as follows:

> "For connectors towards parts that are typed by components defined in a service-oriented layer, the following has to hold: if the source of the connector is typed by an element in a different layer, and if this layer is allowed to access the service-oriented layer, then the target must be a port of a service component providing a service interface."

Since the services return only primitive values and transfer objects, they cannot return references to other objects providing other methods than service. Hence, it is enough to restrict the usage of connectors as mentioned above. *onlyServicesExternallyUsed* is hence defined as follows:

$$\begin{aligned}
&onlyServicesExternallyUsed(x) := \forall g : isAllowedToUseServices(g, x) \rightarrow \\
&\qquad connectToServicesOnly(g, x)
\end{aligned} \tag{6.26}$$

connectToServicesOnly(x, y) describes how allowed connectors must be shaped if instances of components in group x do only use services of components defined in group y. Connectors that are subject to the rule connect a source port p_1 and a target port p_2, defined by components t_1 and t_2, respectively. t_1 is defined in x and t_2 is defined in y. Such a connector is correct if the component defining the target port is a service component, and the target port p_2 provides only service interfaces. This is what the following formula states:

$$\begin{aligned}
&\mathit{connectToServicesOnly}(x,y) := \forall c \forall p_1 \forall p_2 \forall t_1 \forall t_2 : \\
&\qquad \mathit{connectorSource}(c,p_1) \wedge \mathit{connectorTarget}(c,p_2) \wedge \\
&\qquad \mathit{encapsulates}(t_1,p_1) \wedge \mathit{encapsulates}(t_2,p_2) \wedge \\
&\qquad \mathit{inGroup}(t_1,x) \wedge \mathit{inGroup}(t_2,y) \rightarrow \\
&\qquad \mathit{ServiceComponent}(t_2) \wedge (\forall s : \mathit{providedInterface}(p_2,s) \rightarrow \mathit{Service}(s))
\end{aligned} \tag{6.27}$$

It is obvious that the overall rule for ServiceOrientedLayer is an architectural rule. To show that the statement is not local, let us assume we have a system that satisfies *onlyServicesExternallyUsed*(x) for all service-oriented layers x and all incoming dependencies *isAllowedToUseServices*. For one of those layers, we define a new component with a provided port that does not implement a service interface. Furthermore, we create a component in a layer connected to x via a *isAllowedToUseServices* dependency with a required port. Corresponding new parts typed with the new components are created and connected by a new connector. For the selected x, the statement *onlyServicesExternallyUsed*(x) is not fulfilled anymore. All new elements are additions in the sense of the definitions given in Sec. 5.2, hence, the non-locality of this statement is shown.

Section 7.2 will show the checks of this architectural rule for the reference design model of CoCoME; in Sec. 7.3.2, the checks of this rule for the second modification scenario will be presented.

6.2.3. Event-Driven Architecture

The architectural rules for the event-driven architecture of CoCoME can be summarized informally as follows:

1. Components connect to event channels and communicate by the exchange of events. There are components that publish events, and components that subscribe for events; a component can play both roles at the same time. There is no direct communication between the components using events, neither by synchronous nor asynchronous method invocation.

2. The event channel decouples the components by receiving events from the connected publishers and forwarding them asynchronously to the subscribers.

3. Publishers, subscribers, and event channels have to provide and implement certain interfaces to be able to interact.

The third issue means that the single components have to be properly designed to function as a whole. For this purpose, the design profile contains some stereotypes and constraints for the single parts of an event-driven architecture. Those stereotypes are depicted in Fig. 6.7.

The stereotype SubscriberComponent marks components that are subscribers, or consumers, of events. Since they have to react to events, such components have to provide an interface defining signatures for methods that handle events of different types. Such interfaces are marked

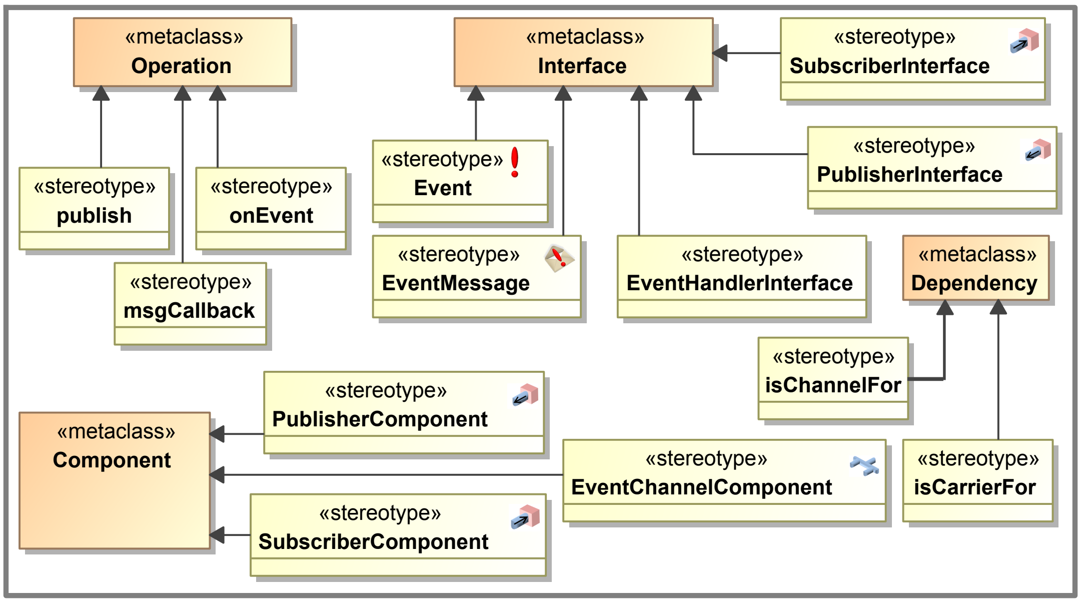

«metaclass» Operation
«metaclass» Interface
«stereotype» SubscriberInterface
«stereotype» PublisherInterface
«stereotype» publish
«stereotype» onEvent
«stereotype» msgCallback
«stereotype» Event
«stereotype» EventMessage
«stereotype» EventHandlerInterface
«metaclass» Dependency
«stereotype» isChannelFor
«stereotype» isCarrierFor
«metaclass» Component
«stereotype» PublisherComponent
«stereotype» EventChannelComponent
«stereotype» SubscriberComponent

Figure 6.7: Stereotypes in the Design Profile to model event publishers and subscribers.

with the stereotype EventHandlerInterface and contain operations stereotyped with onEvent. Those methods are all of similar shape; they have a single parameter typed with an interface stereotyped with Event; it marks interfaces serving as events to publishers, subscribers, and event channels.

Components stereotyped with PublisherComponents create events and sends them to subscribers — possibly directly, if there are point-to-point connections to known subscriber components, or indirectly by an event channel. Publisher components require an interface that tells them by operation signatures how to send an event. Such interfaces are stereotyped with PublisherInterface; the operations used for sending events are stereotyped with publish.

The stereotype EventChannelComponent marks components that can serve as type for actual event channels. Elements that are typed with a component stereotyped this way can furthermore be source of an isChannelFor dependency connecting it with those events that are distributed using the actual event channel.

The mechanism of sending and receiving events is a special application of more general messaging systems for the asynchronous point-to-point or broadcast exchange of messages. These do not necessarily need to transport information about events but can also contain simple text messages, files, etc. For this reason, there are separate stereotypes Message and msgCallback that reflect the more general mechanisms of "off-the-shelf" messaging frameworks that can be utilized for event-driven architectures. msgCallback marks interfaces of messaging frameworks that subscribers must implement as callback; this way the framework can forward messages/events to subscribers. Message is used to mark framework interfaces that serve as "transport vehicle" for events; this means that such interfaces have members which can potentially contain event objects, i.e. instances of Event stereotyped interfaces.

These concepts are defined in the design profile and constitute design rules for component-based systems as it will be shown later in this section.

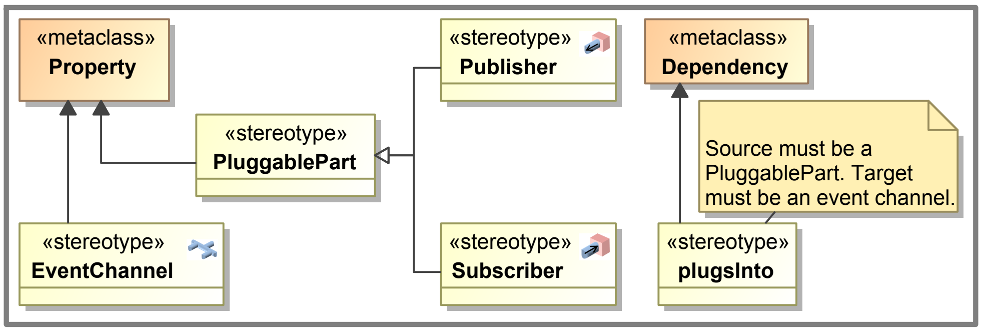


Figure 6.8: Stereotypes in the architecture profile to model event channel architectures.

In the architecture profile, the overall shape of an event-based system or subsystem is restricted by some stereotypes constituting architectural rules; the stereotypes are depicted in Fig. 6.8. Publisher and Subscriber mark parts of system configurations that are publisher components and subscriber components, respectively. EventChannel stereotypes parts that play the role of an event channel. The stereotype plugsInto connects subscribers/publishers with event channels.

The extensional transformations for all elements of the architecture and design profiles are straightforward and similar to the introduced elements of the profiles; a model element elem stereotyped with <Stereotype> is transformed into the statement $Stereotype(e_{elem})$. For the intensional statements, we start with the design rules, i.e. the intensional statements for elements of the design profile.

As mentioned above, we distinguish between events and messages that carry events between publishers and subscribers or event channels. There are no further design rules for events, but we formalize that an event message must contain a member which can be used to carry the event. This means that there must be a member that has a type being a supertype for all events that the message should be able to carry:

$$\begin{aligned} T^{UML,design}(EventMessage) :=& (\{\ldots\}, \{isEventContainer(this)\}) \\ isEventContainer(x) :=& \exists m : (hasMember(x, m) \land \\ & \quad \exists t : (hasType(m, t) \land \\ & \quad\ \forall e : (isCarrierFor(x, e) \rightarrow extends^{+}(e, t)))) \end{aligned} \tag{6.28}$$

The following statements are constructed in similar ways for the triple SubscriberInterface, PublisherInterface, and EventHandlerInterface. The transformation definitions are as follows:

$$\begin{aligned} T^{UML,design}(SubscriberInterface) :=& (\{\ldots\}, \{isValidSubscriberIf(this)\}) \\ T^{UML,design}(PublisherInterface) :=& (\{\ldots\}, \{isValidPublisherIf(this)\}) \\ T^{UML,design}(EventHandlerInterface) :=& (\{\ldots\}, \{isValidEventHandlerIf(this)\}) \end{aligned} \tag{6.29}$$

whereas the statements expressing the validity of a kind of interfaces require the existence of signatures of a defined shape. For subscriber interfaces, the $isValidSubscriberIf(x)$ is defined as follows:

$$\begin{aligned} isValidSubscriberIf(x) :=& SubscriberInterface(x) \land \\ & \exists s : (hasSignature(x, s) \land onMsgCallback(s)) \end{aligned} \tag{6.30}$$

These statements state that a subscriber interface must at least define a message callback signature which must have at least a parameter to which the message that is published to the subscriber can be passed. Hence, message callbacks are transformed into design rules as follows:

$$\begin{aligned} T^{UML,design}(onMsgCallback) :=& (\{\ldots\}, \{hasMsgCallbackSig(this)\}) \\ hasMsgCallbackSig(x) :=& \exists p \exists m : (hasParameter(x, p) \land hasType(p, m) \land \\ & EventMessage(m) \forall q : (hasParameter(x, q) \rightarrow p = q)) \end{aligned} \tag{6.31}$$

Similar, the *isValidPublisherIf*(x) constrains the shape of publisher interfaces; these are interfaces providing methods to send messages and must fulfil the following rule:

$$\begin{aligned} isValidPublisherIf(x) :=& PublisherInterface(x) \land \\ & \exists s : hasSignature(x, s) \land publish(s) \end{aligned} \tag{6.32}$$

They say that publish signatures must provide a signature which has at least one parameter typed with a message interface. This interface can contain events and can be used to transport them.

$$\begin{aligned} T^{UML,design}(publish) :=& (\{\ldots\}, \{hasPublishSig(this)\}) \\ hasPublishSig(x) :=& \exists p \exists m : (hasParameter(x, p) \land hasType(p, m) \land \\ & EventMessage(m) \forall q : (hasParameter(x, q) \rightarrow p = q)) \end{aligned} \tag{6.33}$$

As the last of the three interfaces, event handler interfaces are considered. These interfaces define signatures for methods that are called to react to events:

$$\begin{aligned} isValidEventHandlerIf(x) :=& EventHandlerInterface(x) \land \\ & \exists s : hasSignature(x, s) \land onEvent(s) \end{aligned} \tag{6.34}$$

According to this rule, onEvent signatures are signatures defining a single parameter typed with an event interface.

$$\begin{aligned} T^{UML,design}(onEvent) :=& (\{\ldots\}, \{hasOnEventSig(this)\}) \\ hasOnEventSig(x) :=& \exists p \exists e : (hasParameter(x, p) \land hasType(p, e) \land \\ & Event(e) \land \forall q : (hasParameter(x, q) \rightarrow p = q)) \end{aligned} \tag{6.35}$$

We are now able to define how components must be shaped to serve as type for instances that are meant to be publishers, subscribers, or event channels. The transformation definitions are as follows:

$$\begin{aligned} T^{UML,design}(SubscriberComponent) :=& (\{\ldots\}, \{isValidSubscriberComp(this)\}) \\ T^{UML,design}(PublisherComponent) :=& (\{\ldots\}, \{isValidPublisherComp(this)\}) \\ T^{UML,design}(EventChannelComponent) :=& (\{\ldots\}, \{isValidEventChannelComp(this)\}) \end{aligned} \tag{6.36}$$

The design rules used in all three definitions are similar to each other. They define which ports a component of a kind must provide and will require, and constrain the implementation provided by the component if necessary. For subscriber components, the statements used in the transformation definition are defined as follows:

$$\begin{aligned}
isValidSubscriberComp(x) :=& \exists s \exists h : (isSubPort(s) \wedge \\
& isHandlerPort(h) \wedge forwardsEvent(s,h)) \\
isSubPort(x) :=& ProvidedPort(x) \wedge \\
& \exists i : (providedInterface(x,i) \wedge SubscriberInterface(i)) \\
isHandlerPort(x) :=& ProvidedPort(x) \wedge \\
& \exists i : (providedInterface(x,i) \wedge \\
& EventHandlerInterface(i))
\end{aligned} \tag{6.37}$$

Subscriber components must have a provided port providing an interface that is also a subscriber interface. Moreover, the component must have a provided port providing an event handler interface. The class that serves as type for the port providing the subscriber interface must realize the msgCallback signature in a certain way; the implementation must call an event handler method with the carried event as actual parameter:

$$\begin{aligned}
forwardsEvent(x,y) :=& \exists c : (hasType(x,c) \wedge \forall m \forall s : \\
& (hasMethod(c,m) \wedge implementsSignature(m,s) \wedge onMsgCallback(s) \rightarrow \\
& \exists i \exists h \exists a \exists e : (MethodInvocation(i) \wedge containsStatement(m,i) \\
& \wedge invokedAt(i,y) \wedge invokedSignature(i,h) \\
& \wedge onEvent(h) \wedge parameterValue(i,a) \\
& \wedge actualParameter(a,e) \wedge (\exists p \exists t \exists t' : \\
& hasParameter(s,p) \wedge hasType(p,t) \wedge EventMessage(t) \wedge \\
& hasMember(t,e) \wedge hasType(e,t') \wedge Event(t')))))
\end{aligned} \tag{6.38}$$

For publisher components, we require that there must be a port whose required interface is a publisher interface:

$$\begin{aligned}
isValidPublisherComp(x) :=& \exists p \exists i : (RequiredPort(p) \wedge \\
& requiredInterface(p,i) \wedge PublisherInterface(i))
\end{aligned} \tag{6.39}$$

Event channel components must have a provided port providing a publisher interface. They must have a required port with a subscriber interface; this port will be used to invoke the callback methods of subscribed components. The according design rule is defined as follows:

$$\begin{aligned}
isValidEventChannelComp(x) :=& \exists s \exists p : (requiresSubPort(s) \land \\
& hasPublishPort(p) \land forwardsMessage(p, s)) \\
requiresSubPort(x) :=& RequiredPort(x) \land \\
& \exists i : (requiredInterface(x, i) \land SubscriberInterface(i)) \\
hasPublishPort(x) :=& ProvidedPort(x) \land \\
& \exists i : (providedInterface(x, i) \land \\
& PublisherInterface(i))
\end{aligned} \tag{6.40}$$

For the class that is used as type of the port providing the publisher interface has to hold that the implementations of publish methods invoke message callbacks at the required subscriber port asynchronously:

$$\begin{aligned}
forwardsMessage(x, y) :=& \exists c : (hasType(x, c) \land \forall m \forall s \forall e : \\
& (hasMethod(c, m) \land implementsSignature(m, s) \land publish(s) \land \\
& hasParameter(s, e) \rightarrow \\
& \exists i \exists cb \exists a \ (AsynchronousInvocation(i) \land containsStatement(m, i) \\
& \land invokedAt(i, y) \land invokedSignature(i, cb) \\
& \land onMsgCallback(cb) \land parameterValue(i, a) \\
& \land actualParameter(a, e))))
\end{aligned} \tag{6.41}$$

The transformation rules for the architecture profile add the non-local statements that system parts connected to the same event channel may never communicate directly with each other. In addition, they state the fact that parts have to be properly typed if typed at all. This is concluded by the following transformation definitions:

$$\begin{aligned}
T^{UML,arch}(Subscriber) :=& (\{\ldots\}, \{hasSubscriberType(this)\}) \\
T^{UML,arch}(Publisher) :=& (\{\ldots\}, \{hasPublisherType(this)\}) \\
T^{UML,arch}(EventChannel) :=& (\{\ldots\}, \{hasECType(this), noBypassingOfChannel(this)\})
\end{aligned} \tag{6.42}$$

whereas the single *has<X>Type* statements are defined analogously and state the required typing of parts:

$$\begin{aligned}
hasSubscriberType(x) :=& \forall y : (hasType(x, y) \rightarrow SubscriberComponent(y)) \\
hasPublisherType(x) :=& \forall y : (hasType(x, y) \rightarrow PublisherComponent(y)) \\
hasECType(x) :=& \forall y : (hasType(x, y) \rightarrow EventChannelComponent(y))
\end{aligned} \tag{6.43}$$

The statement *noBypassingOfChannel*(x) expresses that communication between components plugged into a channel has to pass through the channel, and that bypassing is not allowed:

$$\begin{aligned}
&\mathit{noBypassingOfChannel}(x) := \neg\,(\exists c \exists p \exists q : \mathit{sourceContext}(c,p)\wedge\\
&\quad \mathit{targetContext}(c,q) \wedge \mathit{plugsInto}(p,x) \wedge \mathit{plugsInto}(q,x) \wedge \exists q' \exists i \exists j : (\mathit{Event}(j)\\
&\quad \mathit{connectorTarget}(c,q') \wedge \mathit{providedInterface}(q',i) \wedge \mathit{dependsOn}_4(i,j)\wedge\\
&\quad \mathit{isChannelFor}(x,j)))
\end{aligned} \tag{6.44}$$

It states that it is not allowed to introduce a connector that connects two components plugged into the same event channel if the interface of the target port makes use of events in its signatures (see $dependsOn_4$, (6.9)). These rules will be used to check the cash desk subsystem of CoCoME in Sec. 7.2 for architectural compliance and to execute the checks in the third modification scenario (see. Sec. 7.3.3).

6.3. Transformation of UML Models into Logical Statements

The previous section introduced architectural rules for UML models to which the architecture profile or the design profile is applied. In this section, the transformation of UML models into extensional τ_{CBSD}-statements will be described. This means that it will be elaborated how UML models are actually transformed into logical formulae stating which elements need to be present in a system to conform to the model (see Sec. 5.5.1).

More formally, in this section we will define T^{UML} according to the terminology of Sec. 5.5.1, whereas the second component, denoting the set of resulting intensional statements is always the empty set for standard UML. As introduced in Sec. 6.2, intensional statements come into play when the architecture or design profile is applied. This reflects the observation, that UML "as it is" defines extensional statements about systems only [EHK06].

Since there are many similarities in terminology and understanding of concepts between UML and τ_{CBSD}, the transformation is straightforward in most cases. In many cases UML is not as strict as τ_{CBSD} or the corresponding axiom systems Φ_{CBSD}; for example, classes do not necessarily need to be encapsulated in components. Because of those similarities, a formal description of the transformation as in Sec. 5.5.1 will be omitted. Instead, the transformation will be described by example and illustrated informally.

The following subsection focus on the static model aspects that are required to describe component-based systems on the structural level as mentioned in Sec. 4.1.3. For this purpose, class diagrams, component diagrams, and composite structure diagrams need to be transformed. Section 6.3.2 will afterwards introduce the transformation definition for the relevant elements of UML sequence diagrams as required to specify the behaviour of systems as introduced in Sec. 4.1.3.

6.3.1. Transformation of Structural Model Elements

To design the static structure of a component-based system, the modeller can use the three mentioned UML diagram types; component diagrams model components/systems as a black-

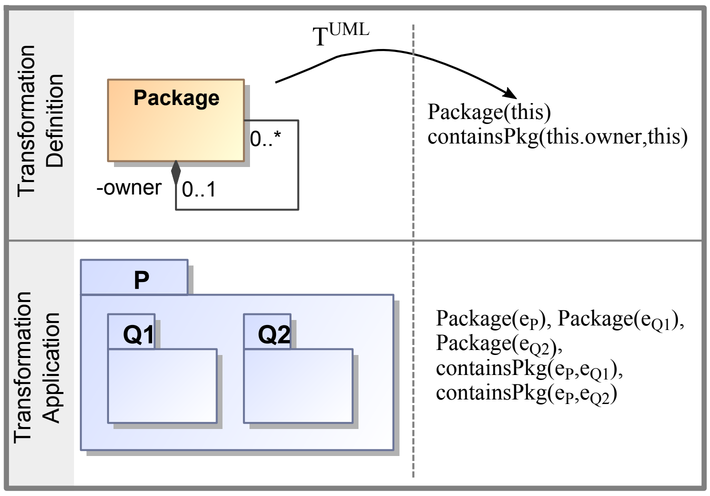

Transformation Definition
T^UML
Package
0..*
-owner
0..1
Package(this)
containsPkg(this.owner,this)
Transformation Application
P
Q1
Q2
Package(e_P), Package(e_Q1),
Package(e_Q2),
containsPkg(e_P,e_Q1),
containsPkg(e_P,e_Q2)

Figure 6.9: Transformation of packages in UML models.

box; composite structure diagrams provide a white-box view on systems and components; class diagrams are used to model details of interfaces and classes. Hence, the most relevant elements of the UML meta model for this section are the three mentioned *classifiers* and some related concepts.

Packages

Packages are used in UML to structure the overall model in separate namespaces. They have basically the same purpose and the same properties like *Package* as relation symbol of τ_{CBSD}. A UML model itself is also a special kind of package, since the meta model element Model is a specialization of Package[1]. Figure 6.9 illustrates how instances of Package are transformed into statements. It illustrates in the upper part the transformation definition of UML meta model elements to extensional logic statements. In the lower part, an example demonstrates the application of the transformation definition.

Every package in a model is transformed into a statement expressing that in a system this package has to be present, whereas the unique identity of the package represented as the constant symbol pointing to that identity (see Sec. 5.5.1). Except the root model package, every package is owned by a package which is defined in the UML meta model by the meta association end owner for instances of Element (see [Obj10b], p. 25). Every appearance of the relationship between an owning package and an owned package is mapped to *containsPkg* $\in \tau_{CBSD}$.

Classifiers

Classifier is the abstract superclass in the UML meta model to encapsulate the commonalities that complex and instantiable types have. A classifier describes "a set of instances that

[1]To keep terms of UML and τ_{CBSD} distinguishable, we will use *italic* font to refer to an element of τ_{CBSD}, and sans-serif to refer to UML meta model elements.

have features in common" ([Obj10b], p. 53). To specify common features, classifiers can define (among others) attributes for structural commonalities and methods for behavioural commonalities of instances. Three specializations of the meta class Classifier are of special interest for the transformation definition mapping UML to τ_{CBSD}-statements: components, interfaces, and classes.

Components in UML have some important properties that differ from features of components in τ_{CBSD}/Φ_{CBSD}:

- As a specialization of Classifier, instances of Component can participate in generalization hierarchies, i.e. components can be specialized and can inherit from each other. Inheritance in τ_{CBSD} is only possible between interfaces and classes, as described in Sec. 5.3.2.
- Just like any other classifier, instances of Component can specify members. This means that components can define attributes and can be connected by associations. Both is not permitted in τ_{CBSD}, attributes and associations, which are both mapped to members, must be defined for interfaces and classes, components can only be connected by connectors. The internal structure of a component can only be defined by parts.
- In UML, the behaviour of components can also be defined and realized by the component itself, comparable to classes. This means, components can specify methods and provide implementation for them. In contrast, in τ_{CBSD} the definition of methods is given by attaching provided interfaces to the component, and implementing them by classes which are specified in the component. The component itself does not define behaviour.
- In UML, components can be hierarchically composed at type level. This means, that a component can be defined as type in the scope of another component, and can be used only there. In τ_{CBSD}, hierarchical composition of components is only possible in the sense that components can be used as type of another component's parts. At type level, only classes can be used to define component-local types.

As explained in Sec. 5.3, the concepts of components and classes are not precisely separated in UML — a fact that manifests in the weaker constraints regarding components in UML as described above. It is hence necessary for the transformation of UML into τ_{CBSD}-statements to deal with models that exploit the weaker constraints and define components with features that are not possible in τ_{CBSD}. e.g. components that define attributes. In the cases above, and also analogous in subsequent transformation definitions, transformations will be defined that way that components are transformed if and only if the stronger constraint of τ_{CBSD} would analogously hold in the UML model. Fig. 6.10 depicts the transformation definitions for components[2].

Components and interfaces in UML are specializations of PackageableElement, the meta class of all elements that can be contained in packages. The relationship between a packageable

[2]Figure 6.10 depicts only a partial transformation definition of Component, as far as the definition of related meta model elements have already been introduced. For example, the definition of the provided ports is omitted here and will be added later (see Fig. 6.17).

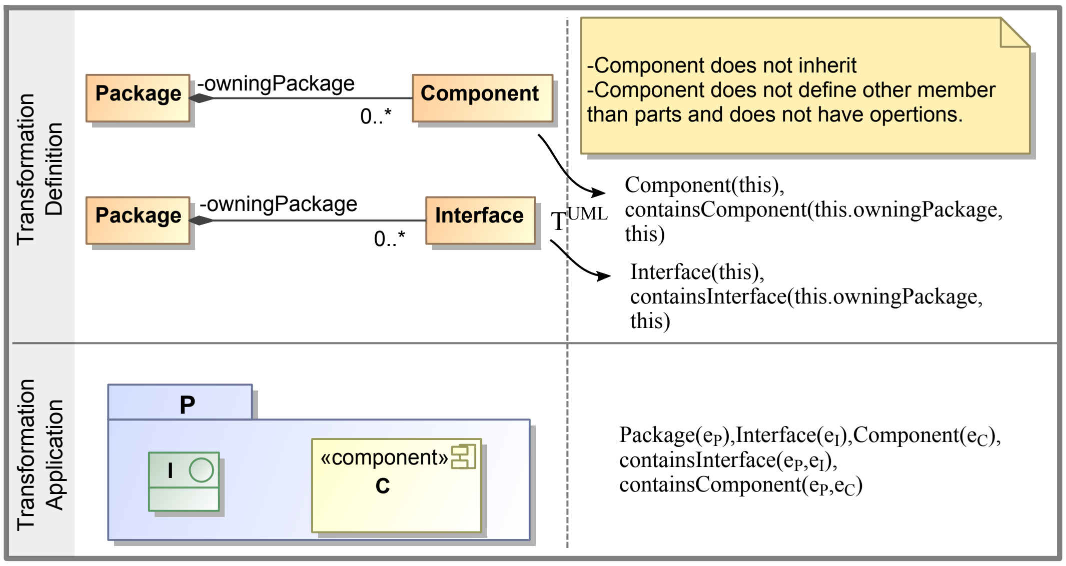

Transformation Definition
Package
-owningPackage
0..*
Component
Package
-owningPackage
0..*
Interface
-Component does not inherit
-Component does not define other member than parts and does not have opertions.
T^UML
Component(this), containsComponent(this.owningPackage, this)
Interface(this), containsInterface(this.owningPackage, this)
Transformation Application
P
I
«component»
C
Package(eP),Interface(eI),Component(eC), containsInterface(eP,eI), containsComponent(eP,eC)

Figure 6.10: Transformation of UML components and interfaces as elements contained in packages.

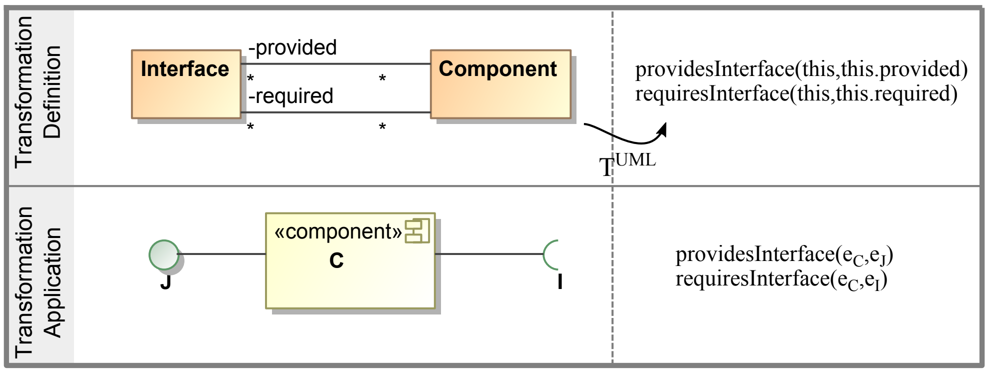

Transformation Definition
Interface
-provided
*
*
-required
*
*
Component
providesInterface(this,this.provided)
requiresInterface(this,this.required)
T^UML
Transformation Application
J
«component»
C
I
providesInterface(eC,eJ)
requiresInterface(eC,eI)

Figure 6.11: Transformation of provides and requires relationships.

element and its containing package is modelled by the meta association depicted in the upper left part of Fig. 6.10, the containing package is referenced by the association end owningPackage. The transformation definition states that for each instance of Component the extensional statement $Component(this)$ is generated, as well as $containsComponent(this.owningPackage, this)$. The same is done analogously for interfaces.

These statements are only generated if the constraints depicted in the note hold, otherwise the result of executing the transformation definition is the empty set.

Components providing or requiring interfaces are modelled in UML very similar to τ_{CBSD}. As Fig. 6.11 shows, those relationships are expressed in the UML meta model by two meta associations, referring to the provided or required interfaces by provided and required, respectively. These are mapped during the transformation of components to corresponding $providesInterface$ and $requiresInterface$ statements.

Classes are used in τ_{CBSD} even more restrictive than in UML. Classes are only local to components in τ_{CBSD} and must be defined in the context of a component (see relation symbol

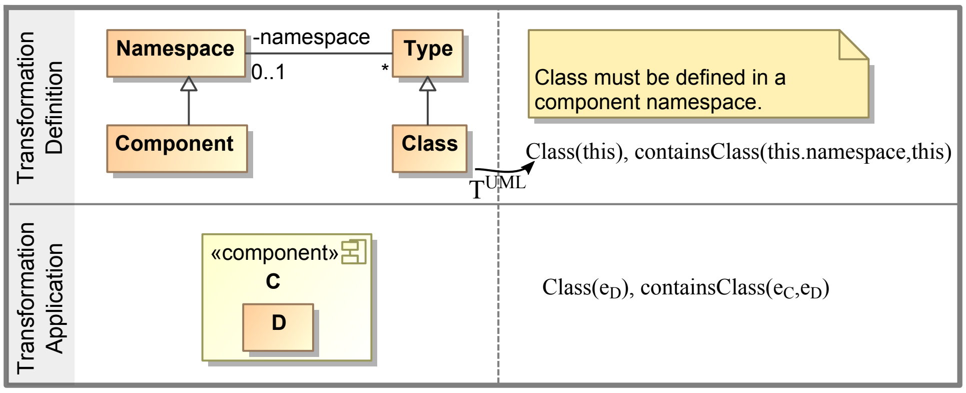

Transformation Definition
Namespace
-namespace
0..1
*
Type
Component
Class
Class must be defined in a component namespace.
Class(this), containsClass(this.namespace,this)
T^UML
Transformation Application
«component»
C
D
Class(e_D), containsClass(e_C,e_D)

Figure 6.12: Transformation of UML classes defined in a component namespace.

containsClass, Sec. 5.3.3). UML as a general purpose modelling language is less restrictive, since it must be possible to model classes without specifying components, for example in object-oriented design models.

The meta class Component inherits from the meta class Namespace which serves as container for named elements[3]. This means that components can be used to define local types like classes. From this follows the first part of the transformation definition for Class as depicted in Fig. 6.12. A class is considered for transformation if its containing namespace is a component, and if this component is transformed into something else than the empty set of statements — if the latter is the case, we could not extensionally claim the existence of the component containing the class. A class with those properties is transformed into a statement corresponding to the definitions $Class(this)$ and $containsClass(this.namespace, this)$. Consequently, the class D inside the component C in the example of Fig. 6.12 is transformed as depicted.

Properties of Classifiers

The UML meta model defines so-called *Features* to summarize model elements that describe structural or behavioural characteristics of classifiers. A special variant of a structural feature is the meta class Property. A property instance can represent either an attribute of a classifier or an association end. Furthermore, a property always belongs to an owning classifier. Since Property is a specialization of TypedElement, every attribute or association end can be typed, for example by another classifier or primitive type.

The upper left part of Fig. 6.13 contains the cutout of the UML meta model relevant for the mentioned issues. It describes the transformation of attributes of classes but it is also valid for the transformation definition of interfaces regarding attributes.

The transformation definition for classes generates a $Member(this.attribute)$ statement for every property that is linked to the current instance of a class via the depicted meta association between Property and Classifier. The statement $definesMember(this, this.attribute)$ is also contained in the transformation definition. Since Property indirectly inherits from TypedElement, the type of the property can be determined by navigating this.attr.type —

[3]In fact, also Package as the most prominent kind of namespace is also a specialization of Namespace.

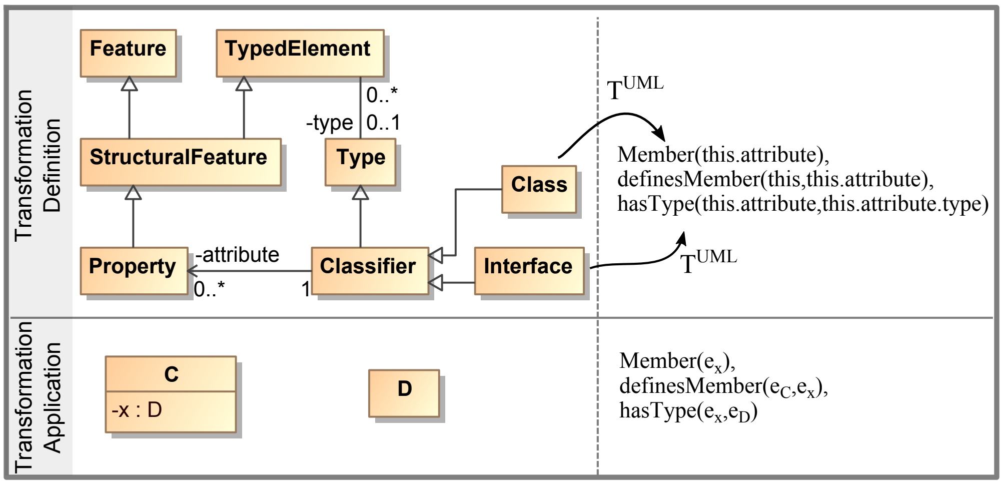


Figure 6.13: Transformation of properties of UML classes or interfaces.

this navigation is used to include the typing of the property in the generated statements as $hasType(this.attribute, this.attribute.type)$. Hence, the example in the lower part of Fig. 6.13 is transformed conforming to the definition above.

As visible by the inheritance hierarchy depicted in the same figure, interfaces share the same required characteristics according properties as classes; hence the transformation definition of interfaces contains the same statements depicted in Fig. 6.13.

The transformation definition concerned with properties so far apply also to association ends owned by one of the associated classifiers. But UML defines also the meta class Association as specialization of Classifier, thus an association can also own the connected ends. This is required in cases when associations have more than two ends or when navigability should be restricted.

To be consistent to the mapping of association ends owned by the participating classifiers to statements claiming the existence of *Member* entities, association ends owned by an association are mapped the same way if the following constraints hold:

- The association has exactly two ends; with more than two ends, it is not clear which associated classifiers should define the member.
- The association end is navigable, i.e. it can be accessed by the classifier on the opposite end of the association.

The transformation definition for associations is illustrated in Fig. 6.14. The ends of an association, that are owned by the association *and* that are navigable, are linked to it via the meta association end navigableOwnedEnd. Hence, if an association end is transformed, a statement *Member*(*this.navigableOwnedEnd*) is created. Its type can be determined by navigating one step further to this.navigableOwnedEnd.type. The classifier that defines this member is that one which can be determined as type of the opposite end of the association: this.navigableOwnedEnd.opposite.type.

Another group of features are behavioural features, contained in the UML meta model as meta class BehaviouralFeature. The specialization Operation is of greatest interest for the

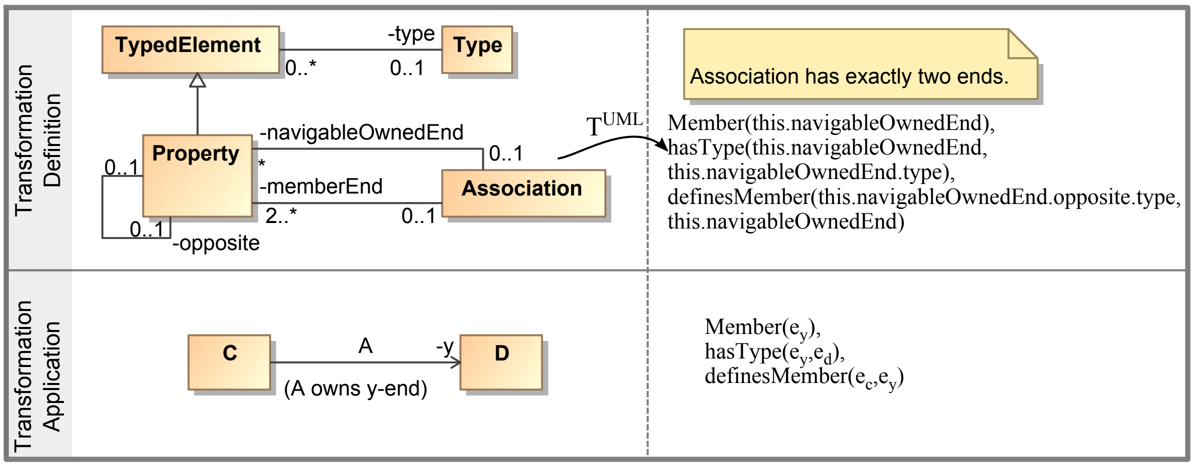

Transformation Definition
TypedElement
-type
Type
0..*
0..1
Association has exactly two ends.
Property
-navigableOwnedEnd
0..1
*
-memberEnd
Association
2..*
0..1
0..1
0..1
-opposite
T^UML
Member(this.navigableOwnedEnd),
hasType(this.navigableOwnedEnd,
this.navigableOwnedEnd.type),
definesMember(this.navigableOwnedEnd.opposite.type,
this.navigableOwnedEnd)
Transformation Application
C
A
-y
D
(A owns y-end)
Member(e_y),
hasType(e_y,e_d),
definesMember(e_c,e_y)

Figure 6.14: Transformation of UML associations; only association ends owned by the association need separate transformation.

transformation of UML models into τ_{CBSD}-statements. Operations are behavioural features of classifiers that specify name, parameters, and types of the behaviour invoked by calling the operation (see [Obj10b], pp. 105). Operations are hence very similar to signatures in τ_{CBSD}.

A behavioural feature, and especially an operation, can be implemented by an instance of Behaviour. A behaviour specifies how the structure and state of instances of the classifier that owns the behavioural features change over time (see [Obj10b], pp. 445). Behaviours associated to operations are hence very similar *MethodBody* entities in τ_{CBSD}.

Figure 6.15 shows how these relationships are modelled in the UML meta model and the transformation into τ_{CBSD}-statements. A BehavioralFeature is owned by a classifier. It defines parameters modelled by the meta class Parameter; parameters are attached to their behavioural feature by the meta association end operation. There are different parameter direction kinds controlling the direction of data flow regarding a single parameter ([Obj10b], pp. 124). The direction kind of a parameter is set by the meta attribute directionKind of Parameter.

The transformation of operations, their parameters, and the implementing behaviours is split into three single parts. The transformation definition for operations creates a statement claiming the existence of a signature entity, correctly typed and assigned to the correct classifier (either interface or class). Parameters are transformed separately if they are not return parameters — these are not required in τ_{CBSD} since the return type is defined by the type of the signature. Statements for the parameter itself, its type, and its assignment to the right signature are created. Furthermore, a *nextParameter* statement is created, setting the previous parameter in the set ownedParameter, if there is any, into relation to the current parameter[4].

To model class hierarchies, the UML meta model defines the meta class Generalization which is connected with two meta associations pointing to the specific class and the general class, respectively. A more specific classifier inherits all the features from the general classifier as far as their visibilities are not private [Obj10b]. The generalization relationships is distinguished from the InterfaceRealization. This relationship connects an interface and a classifier indicating

[4]Since ownedParameter is ordered, it is possible to determine the predecessor of an element in the set.

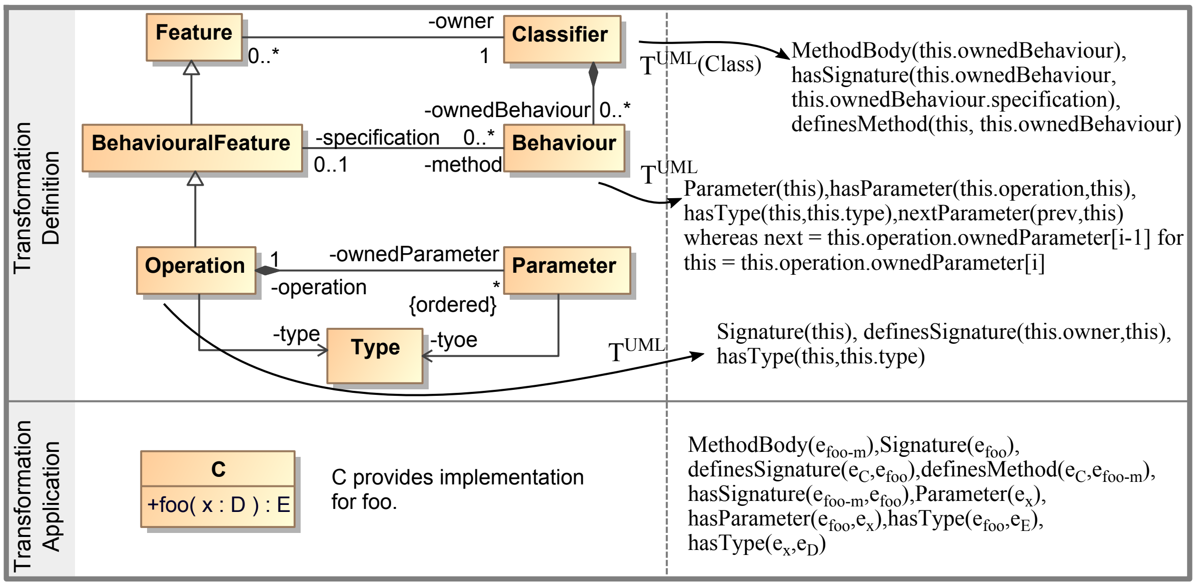

Transformation Definition
Feature
-owner
Classifier
0..*
1
-ownedBehaviour 0..*
BehaviouralFeature
-specification 0..*
Behaviour
0..1
-method
Operation
1
-ownedParameter
Parameter
-operation
*
{ordered}
-type
Type
-tyoe
T^UML(Class)
MethodBody(this.ownedBehaviour), hasSignature(this.ownedBehaviour, this.ownedBehaviour.specification), definesMethod(this, this.ownedBehaviour)
T^UML
Parameter(this),hasParameter(this.operation,this), hasType(this,this.type),nextParameter(prev,this) whereas next = this.operation.ownedParameter[i-1] for this = this.operation.ownedParameter[i]
T^UML
Signature(this), definesSignature(this.owner,this), hasType(this,this.type)
Transformation Application
C
+foo(x : D) : E
C provides implementation for foo.
MethodBody(e_foo-m),Signature(e_foo), definesSignature(e_C,e_foo),definesMethod(e_C,e_foo-m), hasSignature(e_foo-m,e_foo),Parameter(e_x), hasParameter(e_foo,e_x),hasType(e_foo,e_E), hasType(e_x,e_D)

Figure 6.15: Transformation of operations, parameters, and implementing behaviours.

that the classifier implements the interface in the sense that implementing methods are provided for the behavioural features of the interface. Figure 6.16 shows the UML meta model cutout containing both relationships.

The transformation definition for both is very simple; generalizations are only transformed if the connected classifiers are both either classes or both interfaces — this corresponds to the axiom described in Φ_{CBSD} (see Sec. 5.3.3). In this case, the statement is created that claims that this.specific extends this.general, depicted in the upper right part of Fig. 6.16.

The transformation of interface realizations is similar. Here the constraint has to hold that the implementing classifier is a class because otherwise also the interface realization relationship between components and interfaces would match the definition. The created statement is constructed by navigating to this.implementingClassifier and this.contract.

Whether a feature is inherited from a superclass, depends on the visibility of the feature. Inherited features, like attributes and operations, should be added in the generated statements as *inheritsMember* or *inheritsSignature*. For this reason, this.general.attribute is navigated and transformed if the retrieved attribute is not privately visible and transformed into a corresponding *inheritsMember* statement. The same is done for operations, and both are repeatedly defined for interface realizations.

The Structure of Components

UML provides also the meta class Component whose instances will we be transformed into statements claiming the existence of *Component* entities in τ_{CBSD}-structures conforming to the transformed model. There is not a direct counterpart for system configurations (see. Sec. 5.3.4) in UML; instead, a stereotype «system configuration» is introduced. In the following, we will consider the transformation of components without that stereotype only because the transformation to system configuration statements is very similar since the resulting structures

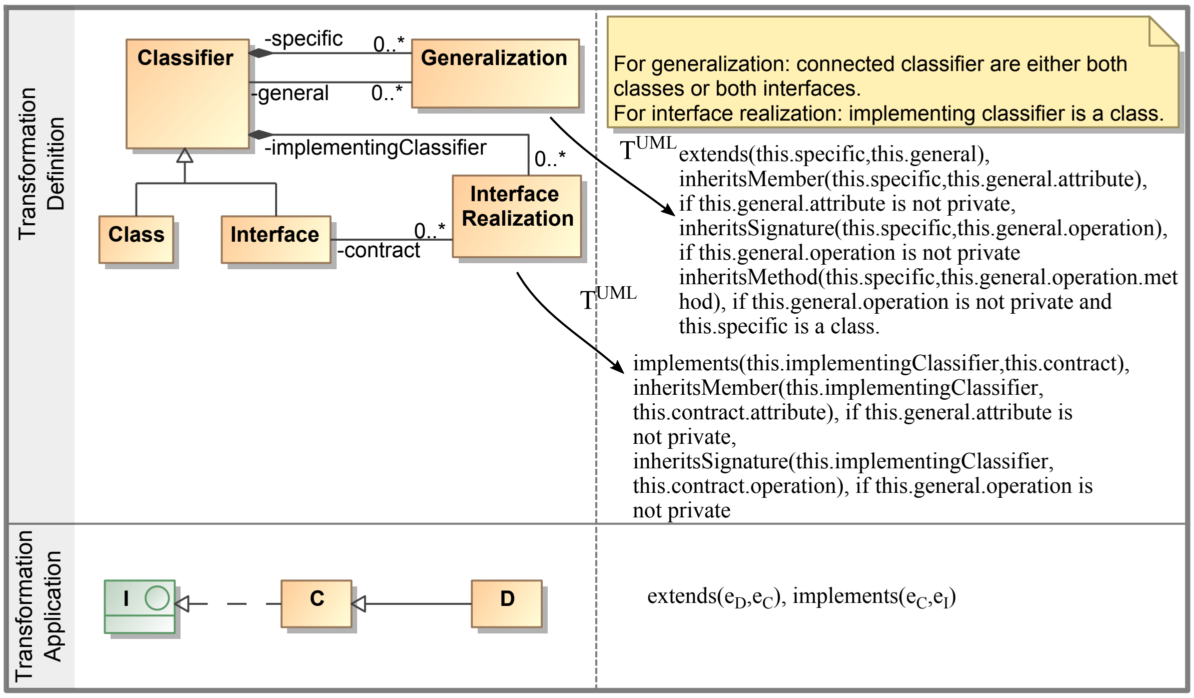

Transformation Definition
Classifier
-specific
0..*
Generalization
-general
0..*
-implementingClassifier
0..*
Class
Interface
0..*
-contract
Interface Realization
For generalization: connected classifier are either both classes or both interfaces.
For interface realization: implementing classifier is a class.
T^UML
extends(this.specific,this.general), inheritsMember(this.specific,this.general.attribute), if this.general.attribute is not private, inheritsSignature(this.specific,this.general.operation), if this.general.operation is not private inheritsMethod(this.specific,this.general.operation.method), if this.general.operation is not private and this.specific is a class.
T^UML
implements(this.implementingClassifier,this.contract), inheritsMember(this.implementingClassifier, this.contract.attribute), if this.general.attribute is not private, inheritsSignature(this.implementingClassifier, this.contract.operation), if this.general.operation is not private
Transformation Application
I
C
D
extends(e_D,e_C), implements(e_C,e_I)

Figure 6.16: Transformation of generalizations and interface realizations.

are very similar, as can be seen in Fig. 5.13, p. 88.

In UML, parts are modelled as instances of the meta class Property that are related to their containing classifier via the meta association end part. This is only possible for classifiers that are also instances of the specialization StructuredClassifier ([Obj10b], p. 169), what is the case for instances of Component which further specializes StructuredClassifier. This means that properties contained in a component via part correspond to those parts in a τ_{CBSD}-structure that are associated to a component by $encapsulates$. To distinguish inner parts from ports (see Sec. 5.3.4), the fact can be considered that UML defines a specialization of Property called Port. In contrast to τ_{CBSD}, UML allows complex ports that require *and* provide an arbitrary number of interfaces at the same time.

Against this background, the transformation of the internal structures of components is defined as depicted in Fig. 6.17. For all properties associated to the component by part, a statement $encapsulates(this.owner, this)$ is created. Each property that is owned by a component and which is not a port, a statement $InnerPart(this)$ is created; if a property is an instance of Port, it is transformed into $ProvidedPort(this)$ or $RequiredPort(this)$, respectively. Statements about type information and the relationships to provided or required interfaces (ports only) are created accordingly.

Connectors in UML have basically the same meaning as connectors of τ_{CBSD}. In contrast to τ_{CBSD}, in which the concept of associations does not exist, connectors can be typed by associations in UML. A connector between two parts of a structured classifier indicates that there can be a link at runtime between the instances playing the roles indicated by the connected parts. The typing by an association defines more precisely which of the properties are linked by

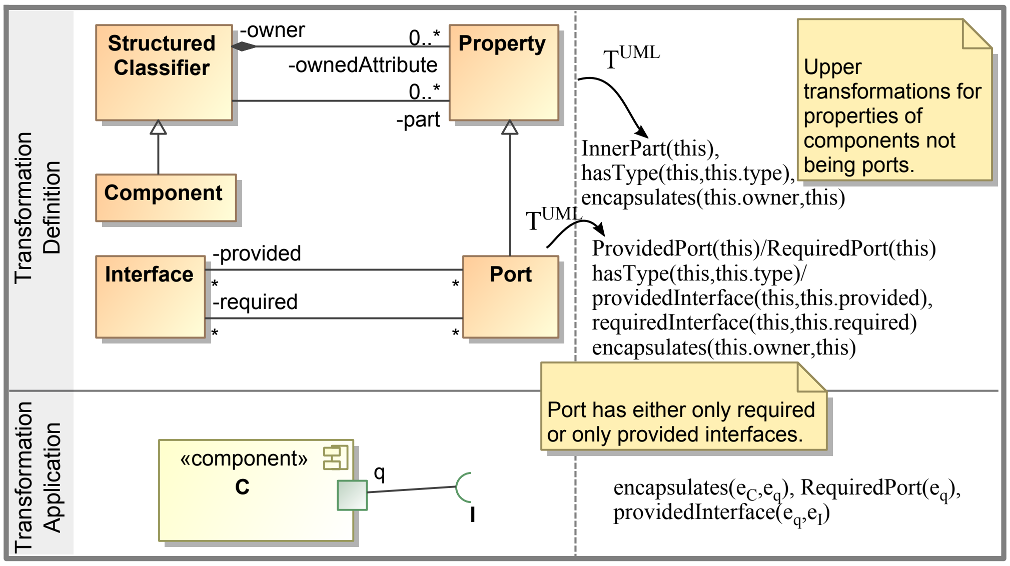

Transformation Definition
Transformation Application
Structured Classifier
-owner
0..*
Property
-ownedAttribute
0..*
-part
Component
Interface
-provided
*
-required
*
Port
*
*
T^UML
InnerPart(this),
hasType(this,this.type),
encapsulates(this.owner,this)
Upper transformations for properties of components not being ports.
T^UML
ProvidedPort(this)/RequiredPort(this)
hasType(this,this.type)/
providedInterface(this,this.provided),
requiredInterface(this,this.required)
encapsulates(this.owner,this)
Port has either only required or only provided interfaces.
«component»
C
q
I
encapsulates(eC,eq), RequiredPort(eq),
providedInterface(eq,eI)

Figure 6.17: Transformation of parts and ports.

the connector. Since associations do not exist in τ_{CBSD}, the corresponding fact is modelled by the *setsMember* relation symbol which defines the member at the source of a connector that is set.

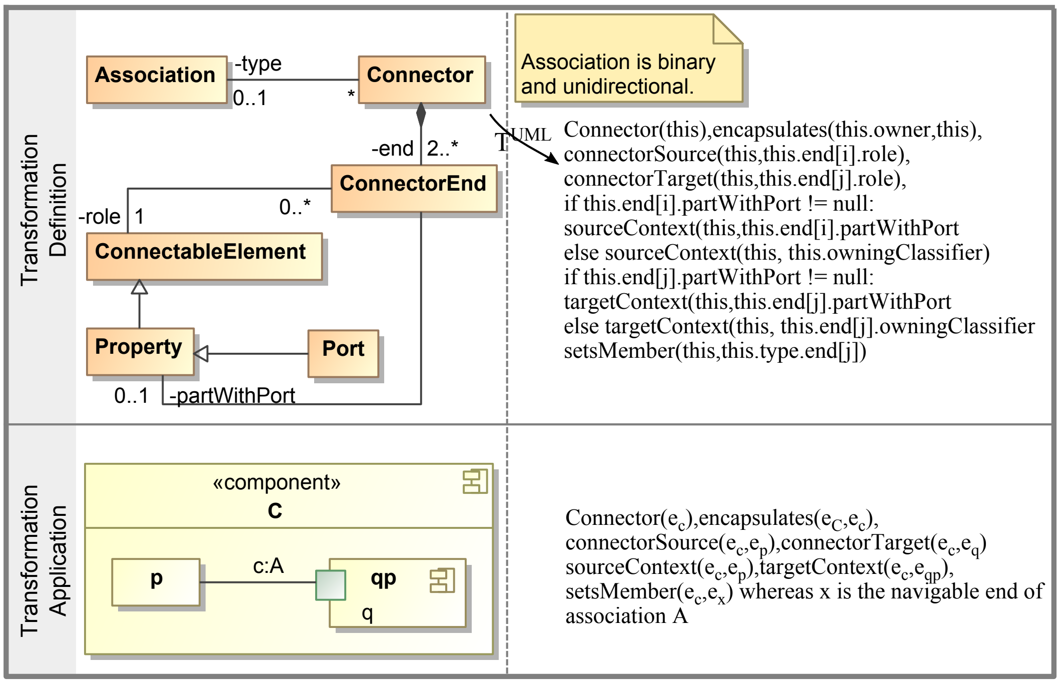
This leads to the following problem of mapping connectors in UML to connectors in τ_{CBSD}. Since connectors set potentially two properties (associations have at least two ends), they cannot be properly mapped to τ_{CBSD}-connectors that set only one member. For that reason, only UML connectors typed by unidirectional associations are transformed in corresponding τ_{CBSD}-statements. The navigable end of the association (connector) in UML corresponds to the member that is set by the connector in a τ_{CBSD}-structure satisfying the statement. Of course, the direction of the typing association has to conform to the direction of connectors in τ_{CBSD} (see Sec. 5.3.4), this means that UML connectors have either

- one end corresponding to a navigable association end, pointing to an inner part, and a non-navigable end pointing at a port providing an interface, *or*
- one end corresponding to a navigable association end, pointing to a port requiring, and a non-navigable end pointing at an inner part, *or*
- both ends pointing at inner parts.

Only in these cases it is possible to generate sensible statements that are satisfiable by structures also satisfying Φ_{CBSD}. Figure 6.18 shows how connectors are transformed into τ_{CBSD} statements.

A connector is transformed into a *Connector* statement if it is typed with an unidirectional association. In the definition it is assumed, that the index j refers to the navigable end in this.type.memberEnd of the set of association ends[5]. The corresponding connector

[5]Remember, only binary associations are considered for transformation.



```
Connector(this),encapsulates(this.owner,this),
connectorSource(this,this.end[i].role),
connectorTarget(this,this.end[j].role),
if this.end[i].partWithPort != null:
sourceContext(this,this.end[i].partWithPort
else sourceContext(this, this.owningClassifier)
if this.end[j].partWithPort != null:
targetContext(this,this.end[j].partWithPort
else targetContext(this, this.end[j].owningClassifier
setsMember(this,this.type.end[j])
```

Connector(e_c),encapsulates(e_C,e_c),
connectorSource(e_c,e_p),connectorTarget(e_c,e_q)
sourceContext(e_c,e_p),targetContext(e_c,e_{qp}),
setsMember(e_c,e_x) whereas x is the navigable end of association A

Figure 6.18: Transformation of UML connectors.

end, this.end[i], is the target of the connector. Index *i* denotes the opposite end. To claim the corresponding context information for the created connector, the meta association end *ConnectorEnd.partWithPort* is used; it refers to that part defining the connected port, if the connector end refers to the port of an inner part. This reference is used to claim the context of the connector in τ_{CBSD}. In case partWithPort is not set, the connector refers to a port of the containing component, and statements about context information are generated accordingly.

6.3.2. Transformation of Behavioural Model Elements

As mentioned in the introduction of this chapter, UML sequence diagrams will be considered for the transformation into τ_{CBSD}-statements as possibility to model system behaviour. Sequence diagrams focus on the interaction aspect of system behaviour, i.e. the exchange of messages between instances of classifiers. Other dynamic aspects, like the changes of instance states, cannot easily be described by sequence diagrams. Since the exchange of messages is focus of the architectural aspects and their rules considered in the case study, only the transformation definition of UML meta model elements relevant for sequence diagrams will be presented.

The central meta model element that is captured by a sequence diagram is Interaction. An interaction is a “unit of behaviour which focuses on the observable exchange of information” between the participants ([Obj10b], p. 498). From this meta model element on, we will discuss the transformation of the details of sequence diagrams.

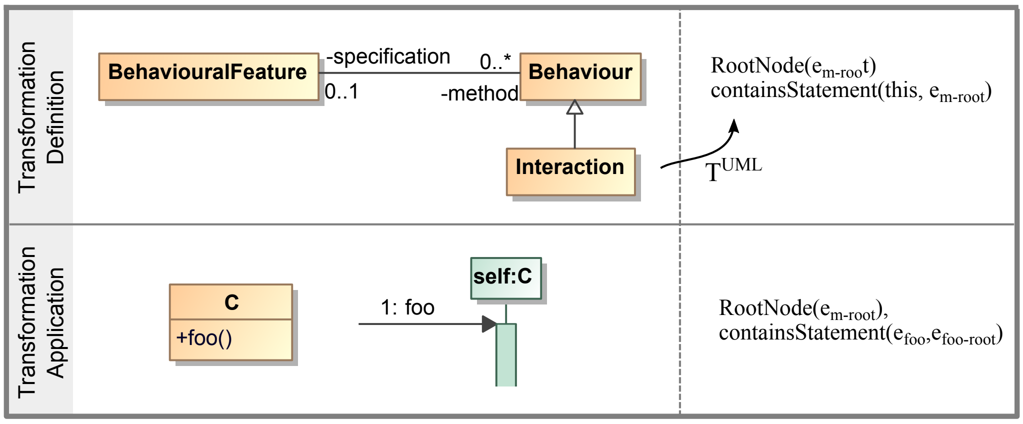


Figure 6.19: Transformation of interactions.

Interactions

Interactions are one of many ways to model behaviour with UML. The meta class Interaction is a specialization of Behaviour. As such, its instances can be assigned as implementations of behavioural features owned by classifiers, as can be seen in Fig. 6.15. Interaction can hence especially be used to specify the behaviour of operations of classes; for a behavioural feature f, the assigned behaviours are referred to by f.method.

A sequence diagram as a whole can be understood as a graphical representation of an interaction. There are certain constraints that have to hold if a sequence diagram, i.e. its underlying interaction, is subject to the transformation.

- The transformation for interactions is only defined for interactions that define the behaviour for operations of classes, since in τ_{CBSD} only classes define behaviour.
- There must be a lifeline named "self" inside the sequence diagram (see[Obj10b], p. 508). The lifeline is typed with the classifier containing the operation that is specified by the interaction.
- There is exactly one incoming message, i.e. a message coming from the diagram frame. Its signature is that of the method the diagram specifies.
- Each messages start or ends at the lifeline named "self".

These constraints refer to meta model elements that will be introduced later in this section, especially Lifeline and Message. For the reader familiar with UML sequence diagrams, Fig. 4.6 (p. 45) is an example in which the constraints hold. The lifeline self is typed by StoreIfImpl which is a class; this way, the implicitly containing classifier is graphically visualized. The incoming message annotated with queryLowStockItems shows which method is depicted by the diagram. Every further message starts or ends at the lifeline self.

Figure 6.19 depicts the transformation definition for Interaction.

The transformation definition for Interaction creates a $MethodBody$ statement; moreover, an $implementsSignature$ statement claims that the method body implements that signature given

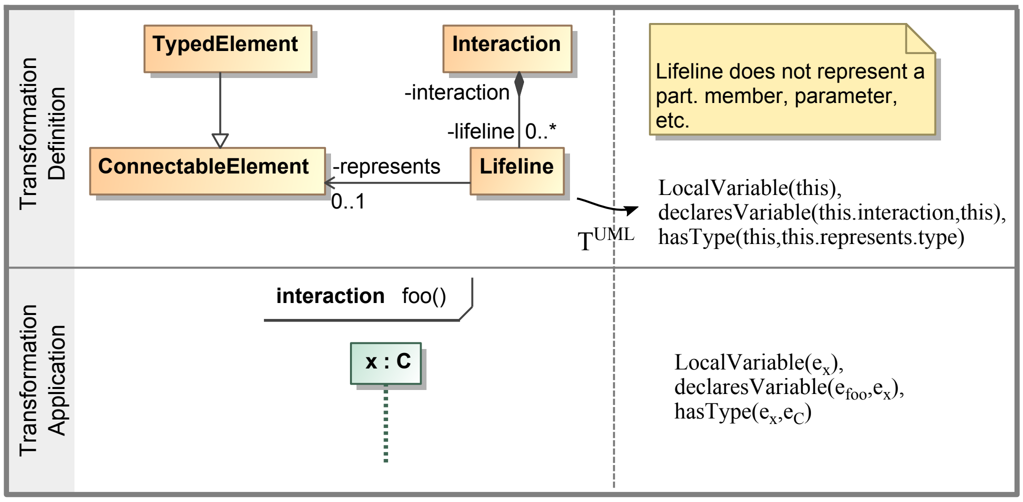

Transformation Definition
TypedElement
Interaction
-interaction
-lifeline 0..*
ConnectableElement
-represents
0..1
Lifeline
Lifeline does not represent a part. member, parameter, etc.
T^{UML}
LocalVariable(this), declaresVariable(this.interaction,this), hasType(this,this.represents.type)
Transformation Application
interaction foo()
x : C
LocalVariable(e_x), declaresVariable(e_{foo},e_x), hasType(e_x,e_C)

Figure 6.20: Transformation of lifelines.

by the implemented behavioural feature. Furthermore, the transformation of Interaction adds a statement for the root node of the call graph that will be generated for the method body. The example depicted in Fig. 6.19 shows that the interaction underlying the diagram is transformed into a root node that is assigned to the method body that is generated for the containing class. In the following, we will refer to this root node entity as e_{m-root} for a signature named m.

Lifelines

Lifelines are the sequence diagrams elements that depict the participants of an interaction. As such they represent instances at runtime that interact with each other. This means that anything in the model that is a placeholder for instances can be used as lifeline — in our case, this can be a part or port of the surrounding component, a member of the class for which the interaction specifies an implementation, or a local variable.

The upper left part of Fig. 6.20 shows the meta model excerpt relevant for the transformation of lifelines. Lifelines are named elements. The part, property, etc., that a lifeline represents, is connected by Lifeline.represents which is an instance of ConnectableElement, a subclass of TypedElement. The interaction, i.e. the actual diagram, the lifeline is contained in, is associated by Lifeline.interaction.

If a lifeline represents a part or member of another classifier, it will not be transformed by the application of $T^{UML}(Lifeline)$. In this case, it represents a part (or port) of a component or a member of a class; in both cases, a different transformation definition already generated a statement claiming the existence of a corresponding reference variable. However, since Behavior is a subclass of Class itself, it can own structural features — such features basically represent local variables. Hence, this relationship is used to define a precondition for the transformation definition of Lifeline; it only applies if the owner of the represented property is the interaction itself.

The generated statements in this case are straightforward. A local variable is claimed, defined in the method body representing the interaction. The required type of the variable is claimed by using this.represents.type as argument for a *hasType* statement.

Messages

Messages as instances of the UML meta class Message are the way how the exchange of information is modelled in sequence diagrams. Of special interest are messages indicating the creation and destruction of instances, the call of operations, and the return from operation calls. The different forms of messages can be specified for a particular message by setting the value of the meta attribute Message.messageSort. The upper left part of Fig. 6.21 depicts the relevant part of the UML meta model.

A message is associated to at most two MessageEnd instances, denoting the sending event (MessageEnd.sendEvent) and the receiving event (MessageEnd.receivingEvent) of the message. MessageEnd is an abstract class and has two instantiable specializations; the meta class MessageOccurenceSpecification represents message ends that are located on lifelines; message ends that are located on the diagram frame, i.e. ends that indicate some event not assigned to a lifeline, are captured by the meta class Gate which is not depicted in the diagram.

The shape of messages that are affected by the transformation definitions is restricted, as mentioned above. Sequence diagrams as used in CoCoME model the implementation of methods comparable to method bodies in object-oriented programming languages. Every method call, creation or destruction method is hence sent by the "self" lifeline. Hence, a corresponding constraint, stating that in these cases sending and receiving ends must be instances of MessageOccurenceSpecification, have to hold.

The creation of objects during an interaction are modelled by a message m with m.messageSort=createMessage. It points to the head of the lifeline representing the newly created instance, as depicted in the lower left section of Fig. 6.21. This kind of message is used to claim the existence of *InstanceCreation* statements in τ_{CBSD}-structures.

For messages of that kind, the transformation is defined as depicted in the upper part of Fig. 6.21. Depending on the kind of classifier, either class or component, an instance creation statement is generated. The instantiated classifier is determined by this.receivingEnd.covered.represents.type; it navigates to the newly created lifeline, from there to the feature it represents, and at last to the feature's type. The property that is assigned by this statements is determined by this.receivingEnd.covered.represents.

The example in the same figure shows the application of this transformation definition. The creation message, depicted as arrow towards the rightmost lifeline, creates a. Assuming that a is an accessible property, the corresponding lifeline is the reference variable to which the new instance is assigned. The type of it is the type of the property/feature the lifeline represents.

The lower transformation definition of Fig. 6.21 transforms delete messages into instance destruction statements. It determines the referenced reference variable (*destructionReference*) the same way like the reference variable for the creation statement above.

Another kind of messages, specified by an according value for the meta attribute Message.messageSort, are synchronous call messages and asynchronous call messages. As before, for those messages, too, the sending event must be located at the "self" lifeline and must point to the lifeline representing the feature at which the operation is called. In case of a synchronous invocation, we have also to consider the corresponding *reply* message modelling

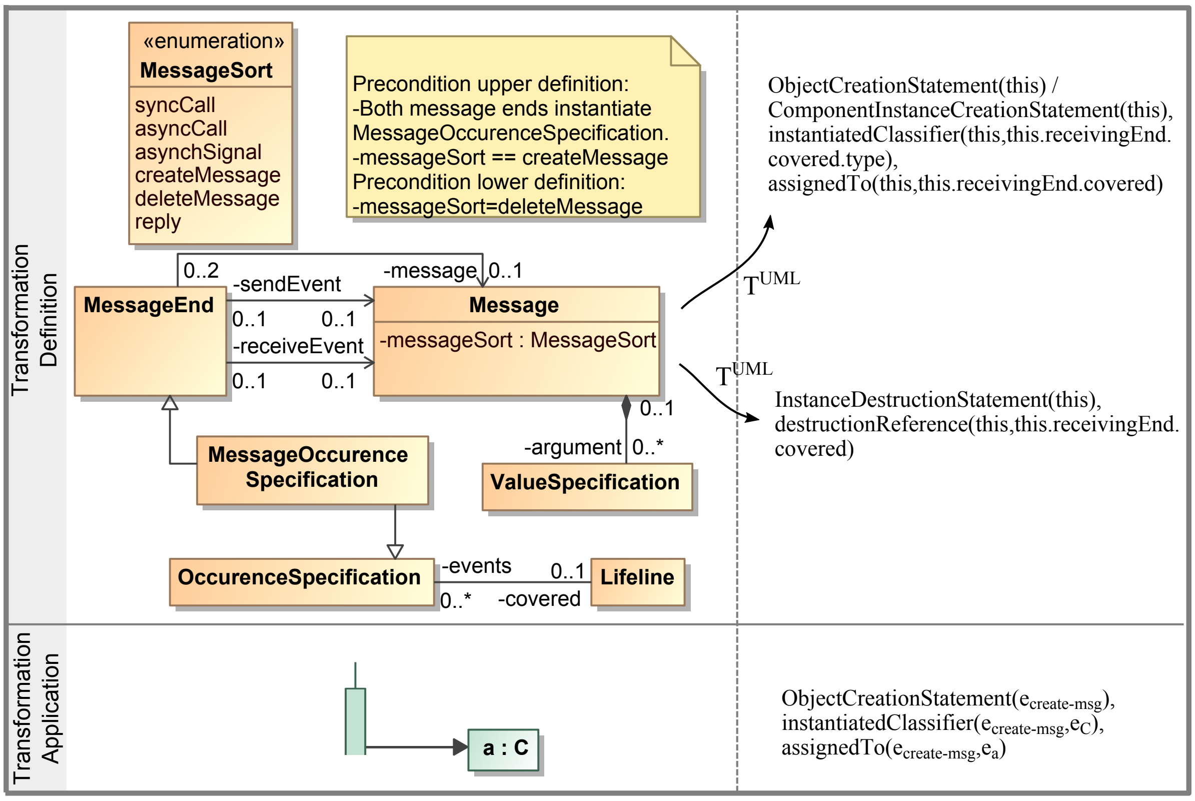

Transformation Definition
«enumeration»
MessageSort
syncCall
asyncCall
asynchSignal
createMessage
deleteMessage
reply
Precondition upper definition:
-Both message ends instantiate MessageOccurenceSpecification.
-messageSort == createMessage
Precondition lower definition:
-messageSort=deleteMessage
ObjectCreationStatement(this) / ComponentInstanceCreationStatement(this), instantiatedClassifier(this,this.receivingEnd.covered.type), assignedTo(this,this.receivingEnd.covered)
T^{UML}
MessageEnd
0..2
-sendEvent
-message
0..1
0..1
0..1
-receiveEvent
0..1
0..1
Message
-messageSort : MessageSort
T^{UML}
InstanceDestructionStatement(this), destructionReference(this,this.receivingEnd.covered)
0..1
-argument
0..*
MessageOccurence Specification
ValueSpecification
OccurenceSpecification
-events
0..1
Lifeline
0..*
-covered
Transformation Application
a : C
ObjectCreationStatement($e_{\text{create-msg}}$), instantiatedClassifier($e_{\text{create-msg}}$,e_C), assignedTo($e_{\text{create-msg}}$,e_a)

Figure 6.21: Transformation of messages creating or destroying lifelines.

the return of the called operation.

This information leads to the transformation definition depicted in Fig. 6.22. A synchronous call message is transformed into a *SynchronousInvocation* statement. The reference of invocation for *invokedAt* can be easily determined by the lifeline containing the receiving event. A message m has the called signature attached by m.signature; however, the signature is in general a NamedElement, not containing about information about parameters and types. However, the constraints of the UML specification refine that relationship and state that, in case m is a call message, the instance referred to by m.signature must be an instance of Operation ([Obj10b], p. 509); parameters and everything that makes up an operation (or signature in τ_{CBSD} terminology) is accessible. The *invokedSignature* can hence easily be generated as depicted.

Moreover, a statement has to be generated for the case that a return value exists. This is passed by a reply message that follows the call message. Since the set of occurrences at a lifeline (lifeline.events) is ordered (see [Obj10b], p. 477), and an object (represented by a lifeline) is blocked until the called operation returns, the event indicating the return must directly follow the sending event. Assuming, that the sending event of the call message is l.events[i], the receiving event of the reply is l.events[i+1]. In the case that the receiving event of the call message is on the same lifeline as the sending event, the reply message is l.events[i+2]. The transformation definition uses these constraints to define the variable next marking the reply receiving event for a call message. The transformation definition for asynchronous call messages is analogous so far.

The generation of the *returnedTo* statement for synchronous call messages refers to the argument specification of next; it is assumed that the argument of the message contains only a single element referring to a lifeline of the interaction.

The last case of messages that are considered for transformation are reply messages starting at the self lifeline and ending at the diagram frame, indicating the return of the method. In this case, too, it is assumed that the list of arguments looks like the reply messages for message calls. This is depicted in Fig. 6.23. The *returnedRef* statement again uses the argument meta property to refer to the returned reference.

The argument property is also used to define the binding of actual and formal parameters for method invocation, which is intentionally omitted here for the sake of brevity. Basically, a constraint in the UML specification ensures that arguments must be given in the same order as parameter of the called operation. Hence, a clear assignment between formal and actual parameter is given and can be used to define corresponding logical statements.

Creation of Control flow graphs

So far, the transformation definition only covers statements that claim the existence of single statement nodes in the control flow graph of a method. We will not go into the details how the overall graph is created but give an informal description.

The transformation of a lifeline create the obligatory root node of the graph. Let us first assume that the interaction for a method does not contain further control flow elements like loops or conditional branches but only a sequence of messages. As already mentioned, the set of events on a single lifeline is ordered; since all call messages, instance creation and delete messages have to be sent by the self lifeline, this order can directly be interpreted as the call

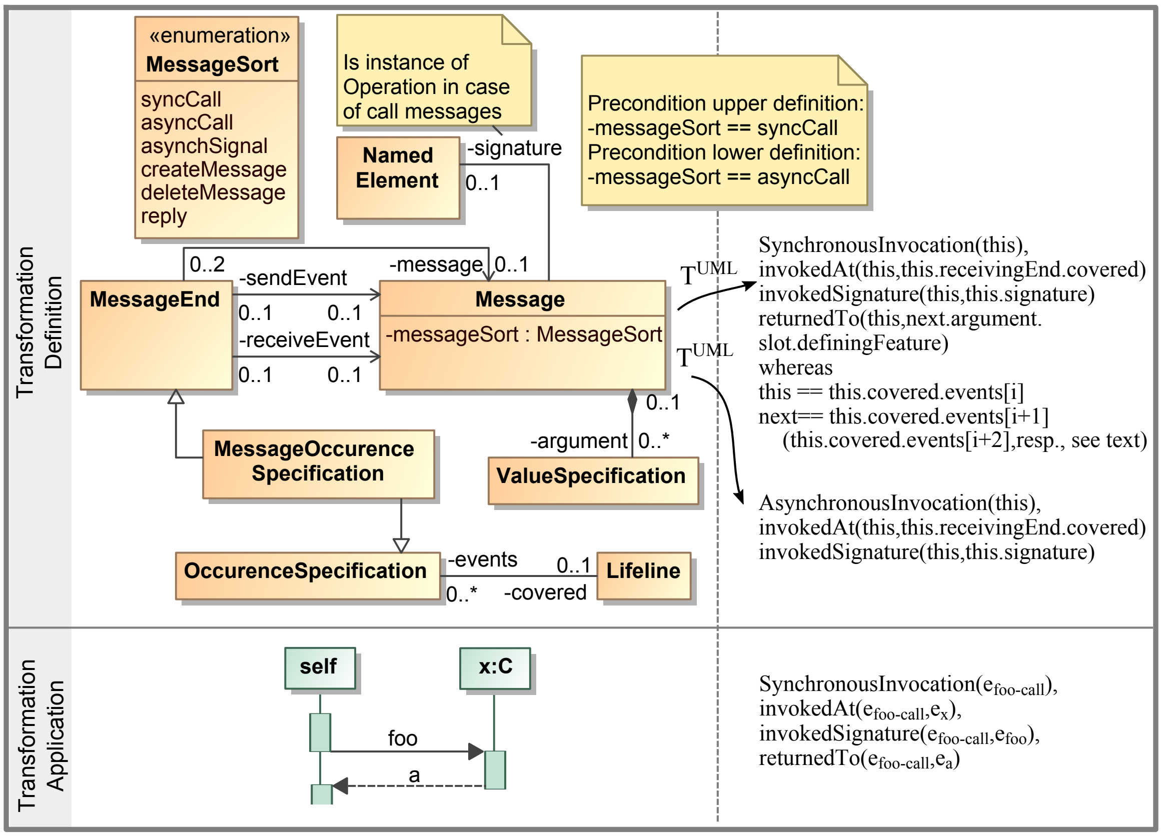

Transformation Definition
«enumeration»
MessageSort
syncCall
asyncCall
asynchSignal
createMessage
deleteMessage
reply
Is instance of Operation in case of call messages
Named Element
-signature
0..1
Precondition upper definition:
-messageSort == syncCall
Precondition lower definition:
-messageSort == asyncCall
0..2
-sendEvent
-message
0..1
MessageEnd
Message
-messageSort : MessageSort
0..1
0..1
-receiveEvent
0..1
0..1
0..1
MessageOccurence Specification
-argument
0..*
ValueSpecification
OccurenceSpecification
-events
0..1
Lifeline
0..*
-covered
T^{UML}
T^{UML}
SynchronousInvocation(this),
invokedAt(this,this.receivingEnd.covered)
invokedSignature(this,this.signature)
returnedTo(this,next.argument.
slot.definingFeature)
whereas
this == this.covered.events[i]
next== this.covered.events[i+1]
(this.covered.events[i+2],resp., see text)
AsynchronousInvocation(this),
invokedAt(this,this.receivingEnd.covered)
invokedSignature(this,this.signature)
Transformation Application
self
x:C
foo
a
SynchronousInvocation($e_{foo\text{-}call}$),
invokedAt($e_{foo\text{-}call}$,e_x),
invokedSignature($e_{foo\text{-}call}$,e_{foo}),
returnedTo($e_{foo\text{-}call}$,e_a)

Figure 6.22: Transformation of messages indicating asynchronous or synchronous calls to operations.

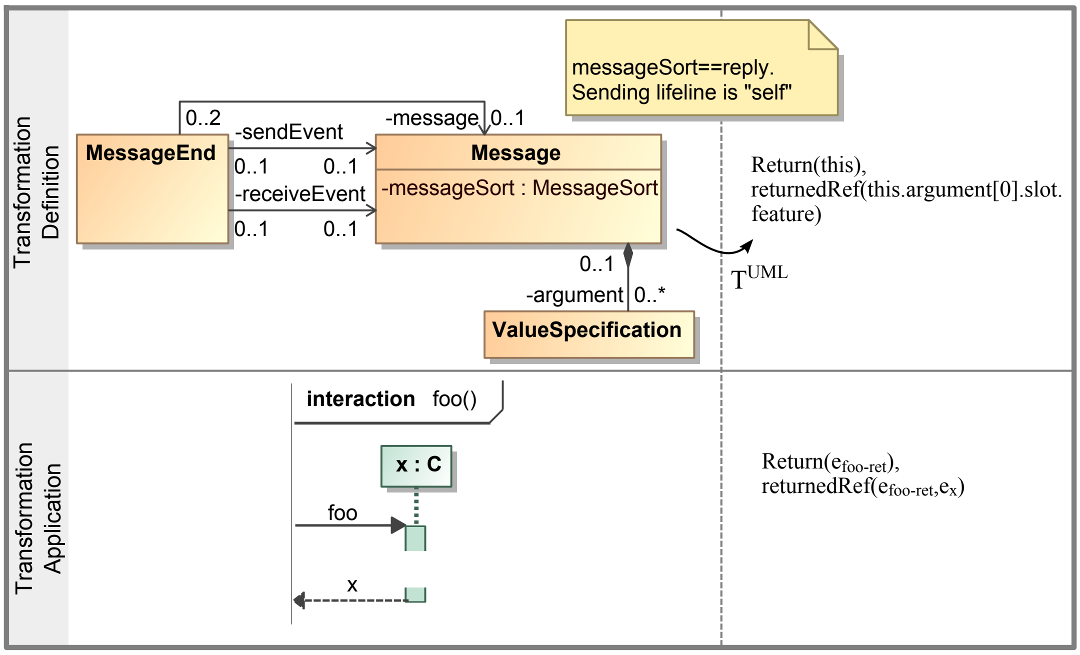

Transformation Definition
messageSort==reply.
Sending lifeline is "self"
0..2
-sendEvent
-message
0..1
MessageEnd
Message
-messageSort : MessageSort
0..1
0..1
-receiveEvent
0..1
0..1
0..1
-argument
0..*
ValueSpecification
T^{UML}
Return(this),
returnedRef(this.argument[0].slot.
feature)
Transformation Application
interaction foo()
x : C
foo
x
Return($e_{foo\text{-}ret}$),
returnedRef($e_{foo\text{-}ret}$,e_x)

Figure 6.23: Transformation of messages indicating the return of methods.

graph of the method; reply messages that follow every call message are ignored in the call graph. The final statement of the interaction, a reply message, is also the final node in the control flow graph.

Sequence diagrams allow also the modelling of loops and alternative blocks, captured by the meta model class CombinedFragment. There is a common superclass of Interaction and CombinedFragment, called InteractionFragment, which is recursively connected to Interaction according to the Composite pattern [GHJV95]. This means that it is possible to build up interaction hierarchically like statement blocks in programming languages; two of them are loops and conditional blocks.

Conditional blocks can easily be reflected in the control flow graphs as introduced. The first nodes of the different paths through the branches are direct successors of the last node before the conditional block. The last node of the paths are direct predecessors of the first node after the conditional blocks.

The actual representation of loop blocks depends, according to the UML specification, on the kind of loop condition for which there are many possibilities like specification of minimal/maximal number of iterations, constraints, etc. If we assume that the condition is a boolean expression, the loop block compares best to “while loops” in programming languages. Formalized as control flow graph, the last node before such a loop is connected to the first node inside the loop (executed if condition is true) and the first node after the loop (executed if condition is false). The node representing the last statement inside the node is followed by the first node of the loop (condition is true after an iteration) and the first node after the loop (last iteration through loop). For the body of the loop, the control flow graph is iteratively constructed according to the rules for sequences, conditional blocks, and loops.

6.4. Summary

The main objective of this chapter has been to provide the transformation definition for languages being able to create architectural models and design models. For this purpose, two issues have been addressed. First, the approach that has been introduced in detail in Chapter 5 has been used to define architectural rules for UML as architecture description language. Two UML profiles have been developed, motivated by the architectural aspects of the CoCoME case study. An architecture profile has been specified containing several stereotypes capturing the actual “architectural” essence of the informally identified aspects; the layering of a system influences the allowed usage relationships; a service-oriented interface permits the usage of certain components only; an event bus may not be bypassed. All of these rules have been proved to be architectural, i.e. intensional and non-local. They have been formalized as logical statements and have been used to specify the transformation definition for UML model elements stereotyped with elements from the architecture profiles.

Parts of the informally described architectural aspects affect also design models because they define intensional but local constraints over component-based systems. The definition of what actually constitutes a service component or a component that is a publisher of events, can be given as design rules. Hence, also a design profile has been introduced. A transformation definition has been given for UML models applying this profile.

The second issue has been the transformation definition for standard UML as foundation for the selected architecture description language (UML + Architecture Profile) as well as foundation for design models (UML + design model). For these purposes, we have seen in this chapter how to transform the static structure modelled by class diagrams, component and composite structure diagrams into extensional τ_{CBSD}-statements. Furthermore, transformation definitions have been given for the relevant elements of UML sequence diagrams.

We are now able to check architectural compliance of UML design models with UML architecture models applying the architecture profile. Given such models for CoCoME, the architectural compliance of the case study can be checked.

Chapter 7.

Architecture Compliance Checking of the Case Study

Contents

A man’s errors are his portals of discovery.

James Joyce

The previous chapter prepared the ground to check the architectural compliance of the CoCoME case study; the architectural rules for the main architecture aspects have been defined, thus it is defined “what the architecture means” for design and implementation; a simple ADL based on UML has been defined, thus the architecture of CoCoME can be modelled; and, last but not least, the transformation of UML has been defined, thus architecture and design model can be transformed into logical statements. Together with the defined operationalisation, the scene is set to check architectural compliance for the CoCoME case study.

This chapter contains the results of checking architectural compliance in the different scenarios; first, the developed reference design model of CoCoME is checked, and the results are evaluated; afterwards, the compliance checks as part of the modification scenarios introduced in Sec. 4.3 are executed.

The chapter is structured as follows. First, the architectural model will be described. In Sec. 7.1.2, the reference design model will be illustrated; the results of checking its compliance with the architecture model are described in Sec. 7.2. Section 7.3 contains the three modification scenarios in which architecture compliance checking is used to detect erroneous evolutions of the system's design model.

7.1. Models of the Trading System

This section presents the architectural model and the reference design model for the CoCoME case study. First, the architectural model, the logical statements including the architectural rules, and a minimal system for the model will be developed. In the second subsection, the reference design model and its transformation into logical statements will be described by example.

7.1.1. Architectural Model

The architecture model of CoCoME is a UML model to which the architecture profile introduced in Sec. 6.2 is applied. This means that all UML language elements and the stereotypes defined in that profile can be used. The complete architectural model is depicted in Fig. 7.1. The illustration can be briefly separated into two areas; the upper part of the figure contains the inventory subsystem architecture; the lower part contains the cash desk subsystem architecture.

The model describing the inventory subsystem architecture illustrates the three layers and their mapping to a package structure as introduces in Sec. 6.2.1. This package structure in this minimal form is also part of the architectural model; the packages contained are populated with components and interfaces later in the design model. Between GUI layer and application layer as well as between application layer and data layer the allowed usage relations are modelled.

Furthermore, the application layer is not simply stereotyped with Layer but with the specializing stereotype ServiceOrientedLayer (see Sec. 6.2.2). This indicates that the GUI layer, connected by a isAllowedToUse relationship is allowed to use the application layer by the use of service interfaces and transfer objects only.

The lower part of Fig. 7.1 models the cash desk system part of the TradingSystem. It contains a system configuration that describes how components may interact via event channels. There are two event channels modelled, according to the description in Sec. 4.1.2. The single device controllers and the cash desk application are modelled by parts encapsulated in the system configuration. They are stereotyped as publishers or subscribers depending on their usage of events (see. Sec. 6.2.3). Note, that these parts are not typed yet. The specific components that are going to by types of that parts will be defined in the design model.

The reference design model of CoCoME, and the modified design models developed in the three modification scenarios, will be checked against this architectural model. The architectural model is transformed into logical statements according to the rules defined in Sec. 6.2 and

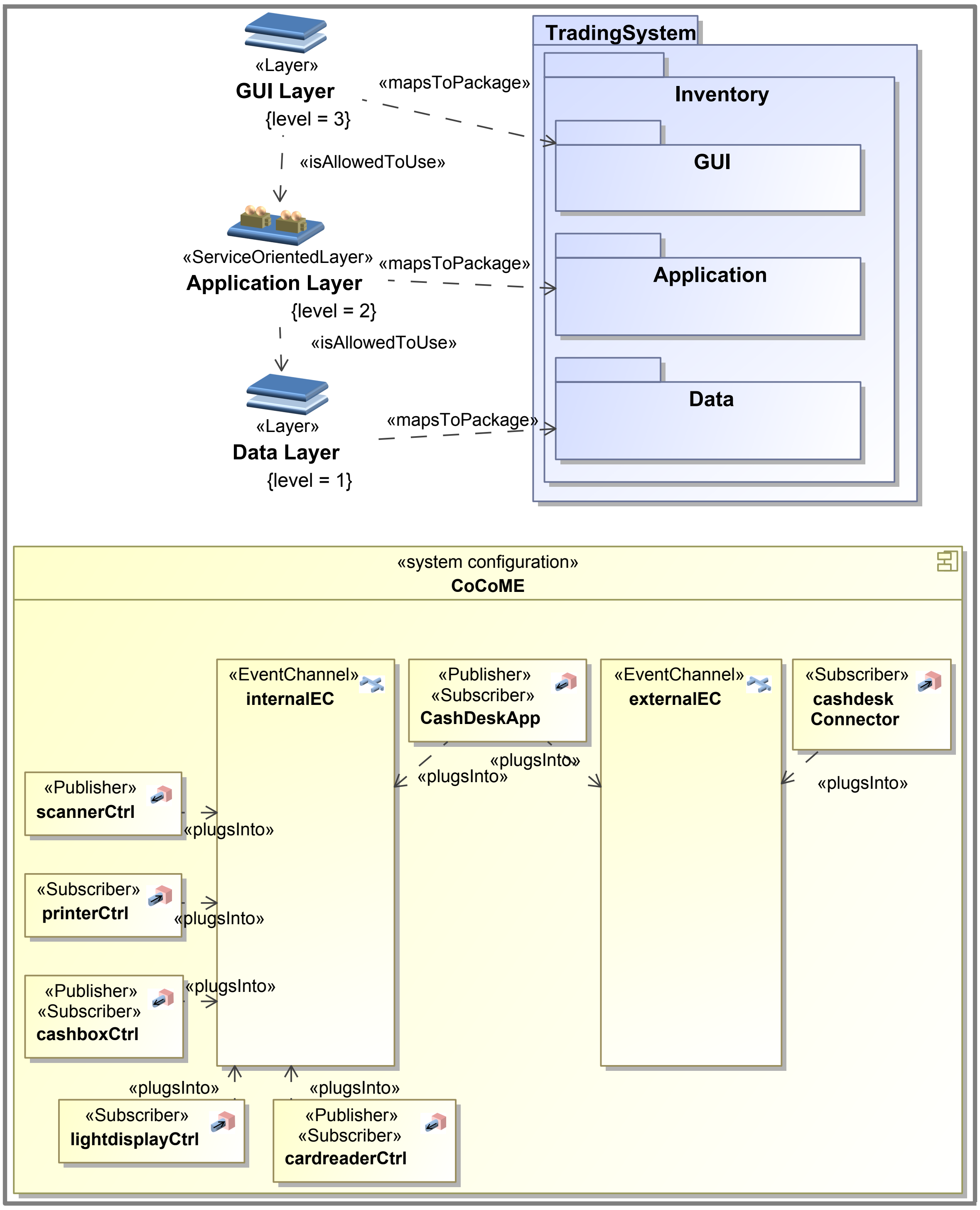


Figure 7.1: Architectural model of CoCoME.

Sec. 6.3. First, we will take a look upon the extensional statements created by the general transformation of UML models described in Sec. 6.3.

The inventory part of the architectural model defines the following extensional statements:

$$\begin{aligned}
\{\, & Layer(e_{GUILayer}), ServiceOrientedLayer(e_{ApplicationLayer}),\\
& Layer(e_{DataLayer}),\\
& isAllowedToUse(e_{GUILayer}, e_{ApplicationLayer}),\\
& isAllowedToUseServices(e_{GUILayer}, e_{ApplicationLayer}),\\
& isAllowedToUse(e_{ApplicationLayer}, e_{DataLayer}),\\
& Package(e_{TradingSystem}), Package(e_{Inventory}), Package(e_{GUI}), Package(e_{Application}),\\
& Package(e_{Data}),\\
& mapsToPackage(e_{GUILayer}, e_{GUI}), mapsToPackage(e_{ApplicationLayer}, e_{Application}),\\
& mapsToPackage(e_{DataLayer}, e_{Data}),\\
& containsPkg(e_{TradingSystem}, e_{Inventory}), containsPkg(e_{Inventory}, e_{GUI}),\\
& containsPkg(e_{Inventory}, e_{Application}), containsPkg(e_{Inventory}, e_{Data})\}
\end{aligned} \tag{7.1}$$

For reasons of clarity, the statements additionally generated for model elements due to generalization hierarchies in the meta model are omitted in the set above and the following. For example, the statements $Group(e_{DataLayer})$ and $Package(e_{DataLayer})$ are also elements of the set because Layer specializes Group which stereotypes Package.

The cash desk system architecture is transformed into the following extensional statements:

$$\begin{aligned}
\{\, & SystemConfiguration(e_{CoCoME}),\\
& EventChannel(e_{internalEC}), EventChannel(e_{externalEC}),\\
& Publisher(e_{scannerCtrl}), Publisher(e_{cashboxCtrl}),\\
& Publisher(e_{cardreaderCtrl}), Publisher(e_{cashDeskApp}),\\
& Subscriber(e_{printerCtrl}), Subscriber(e_{lightdisplayCtrl}),\\
& Subscriber(e_{cardreaderCtrl}), Subscriber(e_{cashDeskApp}), Subscriber(e_{sa}),\\
& plugsInto(e_{scannerCtrl}, e_{internalEC}), plugsInto(e_{printerCtrl}, e_{internalEC}),\\
& plugsInto(e_{cashboxCtrl}, e_{internalEC}), plugsInto(e_{lightdisplayCtrl}, e_{internalEC}),\\
& plugsInto(e_{cardreaderCtrl}, e_{internalEC}), plugsInto(e_{cashdeskApp}, e_{internalEC}),\\
& plugsInto(e_{cashDeskApp}, e_{externalEC}), plugsInto(e_{sa}, e_{externalEC})\}
\end{aligned} \tag{7.2}$$

The created architectural rules are completely defined by the transformation definitions given in the architectural profile. According to those definitions, the Layer elements lead to the following architectural rules:

$$\begin{aligned}
\{ & \neg\, illegalDependenciesFromGroup(e_{GUILayer}),\\
& \neg\, illegalDependenciesFromGroup(e_{ApplicationLayer}),\\
& \neg\, illegalDependenciesFromGroup(e_{DataLayer})\}
\end{aligned} \tag{7.3}$$

Relation symbol	Interpreting relation
$Layer$	$\{e_{GUILayer}, e_{ApplicationLayer}, e_{DataLayer}\}$
$ServiceOrientedLayer$	$\{e_{ApplicationLayer}\}$
$Package$	$\{e_{TradingSystem}, e_{Inventory}, e_{GUI}, e_{Application}, e_{Data},$ $\{e_{GUILayer}, e_{ApplicationLayer}, e_{DataLayer}\}$
$containsPkg$	$\{(e_{TradingSystem}, e_{Inventory}), (e_{Inventory}, e_{GUI}),$ $(e_{Inventory}, e_{Application}), (e_{Inventory}, e_{Data})\}$
$isAllowedToUse$	$\{(e_{GUILayer}, e_{ApplicationLayer}), (e_{ApplicationLayer}, e_{DataLayer})\}$
$isAllowedToUseServices$	$\{(e_{GUILayer}, e_{ApplicationLayer})\}$
$mapsToPackage$	$\{(e_{GUILayer}, e_{GUI}), (e_{ApplicationLayer}, e_{Application}),$ $(e_{DataLayer}, e_{Data})\}$
$EventChannel$	$\{e_{internalEC}, e_{externalEC}\}$
$Publisher$	$\{e_{scannerCtrl}, e_{cashboxCtrl}, e_{cardreaderCtrl}, e_{cashdeskApp}\}$
$Subscriber$	$\{e_{printerCtrl}, e_{lightdisplayCtrl}, e_{cardreaderCtrl}\}$ $\{e_{cashdeskApp}, e_{sa}\}$
$plugsInto$	$\{(e_{scannerCtrl}, e_{internalEC}), (e_{printerCtrl}, e_{internalEC})\}$ $\{(e_{cashboxCtrl}, e_{internalEC}), (e_{lightdisplayCtrl}, e_{internalEC})\}$ $\{(e_{cardreaderCtrl}, e_{internalEC}), (e_{cashdeskApp}, e_{internalEC})\}$ $\{(e_{cashdeskApp}, e_{externalEC}), (e_{sa}, e_{externalEC})\}$

Table 7.1: Minimal system for the architectural model of CoCoME.

Moreover, the fact that the application layer is modelled as a service-oriented layer, implies the following set of rules:

$$\{onlyServicesExternallyUsed(e_{ApplicationLayer})\} \tag{7.4}$$

Finally, the architecture of the cash desk subsystem leads to the following architectural rules

$$\{noBypassingOfChannel(e_{internalEC}), noBypassingOfChannel(e_{externalEC})\} \tag{7.5}$$

Please note that the typing rules regarding subscribers, publishers, and event channels (see Sec. 6.2.3, statement (6.43)) are not contained here because they are not architectural; they are intensional but local. Since the corresponding parts are not typed, they are nevertheless fulfilled in the architecture model.

There is a unique minimal structure of the architectural model since the generated extensional statements are unambiguously factual (see Sec. 5.5.3). The minimal structure for the extensional statements given in expressions (7.1) and (7.2), is defined as concluded in Table 7.1.

The minimal system of the architectural model is required to check compliance of design models or implementation models (see Sec. 5.5.3). It can easily be seen that the minimal structure described in Tab. 7.1 is also a system that conforms to τ_{CBSD} and Φ_{CBSD}. The only elements of τ_{CBSD} that are referenced are *Package*, *containsPkg*, *SystemConfiguration*, *Part* and *configurationPart*. All axioms considering these elements are followed (see App. A).

Furthermore, the model is also intensionally complete. The minimal system for the extensional statements that the model defines, is also a system that satisfies the intensional statements, i.e. the architectural rules. First, the architectural rules of the set (7.3) state that no dependencies with certain properties, defined in detail in Sec. 6.2.1, are allowed. Since there are no dependencies in the sense of the formulae $\mathit{dependsOn}_i$ at all, the statements are obviously true for the minimal system. Second, for the same reasons, rules defined by the service-oriented application layer are fulfilled (see Sec. 6.2.2). The statements regarding the bypassing of event channels are obviously true — there are not any connectors that could bypass one of the event channels.

Concluding, all three assumptions required for compliance checking defined in Sec. 5.5.3 are true for the architectural model of CoCoME. Consequently, it can serve as input to the algorithm to architectural compliance checking.

7.1.2. Design Model

The reference design model of CoCoME consists of a system configuration at the top-level, detailed white- and black-box models of the single components, and sequence diagrams as behaviour descriptions of the methods implementing operations. Since a complete description of the design model would be out of the scope of this chapter, the model will be described only partially in a top-down order. The system configuration is described completely, after that, only a single component will be described in detail. A single method of the implementation of that component will be illustrated.

A more exhaustive description of the reference design model can be found in App. C. The description provides a complete static view on the Trading System and more exemplary behaviour descriptions.

Fig. 7.2 depicts the inventory part of the overall system configuration. It consists of eight components. Three of them support the system functionality to manage a store, two provide reporting functions (see also Sec. 4.1.1). The three store-related components are distributed over the layers, one providing the user interface associated to the functional area, one containing the application logic, and one containing the persistent data model for stores (in contrast to Enterprise which contains the data model relevant for whole enterprises). The layering is visually reflected by the vertical order of the groups of components. The two additional parts provide the technical persistence mechanisms (:Persistence) and the functionality to move goods between stores (:ProductDispatcher).

The types of the parts are components that are contained in a subtree of the overall package hierarchy of the TradingSystem depicted in Fig. C.1 and Fig. C.2 — which is the same like the subtree already modelled in the architectural model (see. Fig. 7.1).

Figure 7.3 depicts those parts of the overall system configuration that model the cash desk part of the system. Some of the ports of the parts are omitted for the sake of clarity as well as the mapping between event channels and the events they forward modelled by isChannelFor; the internal event channel, however, is carrier for every event defined in the system. The structure of the subsystem with the central event channels are clearly visible. There is the internal event channel connecting the cash desk application and the controllers for the single devices. The external event channel connects the cash desk application with the StoreApp component of the inventory system part. In contrast to the architectural model, which contains also a more-coarse

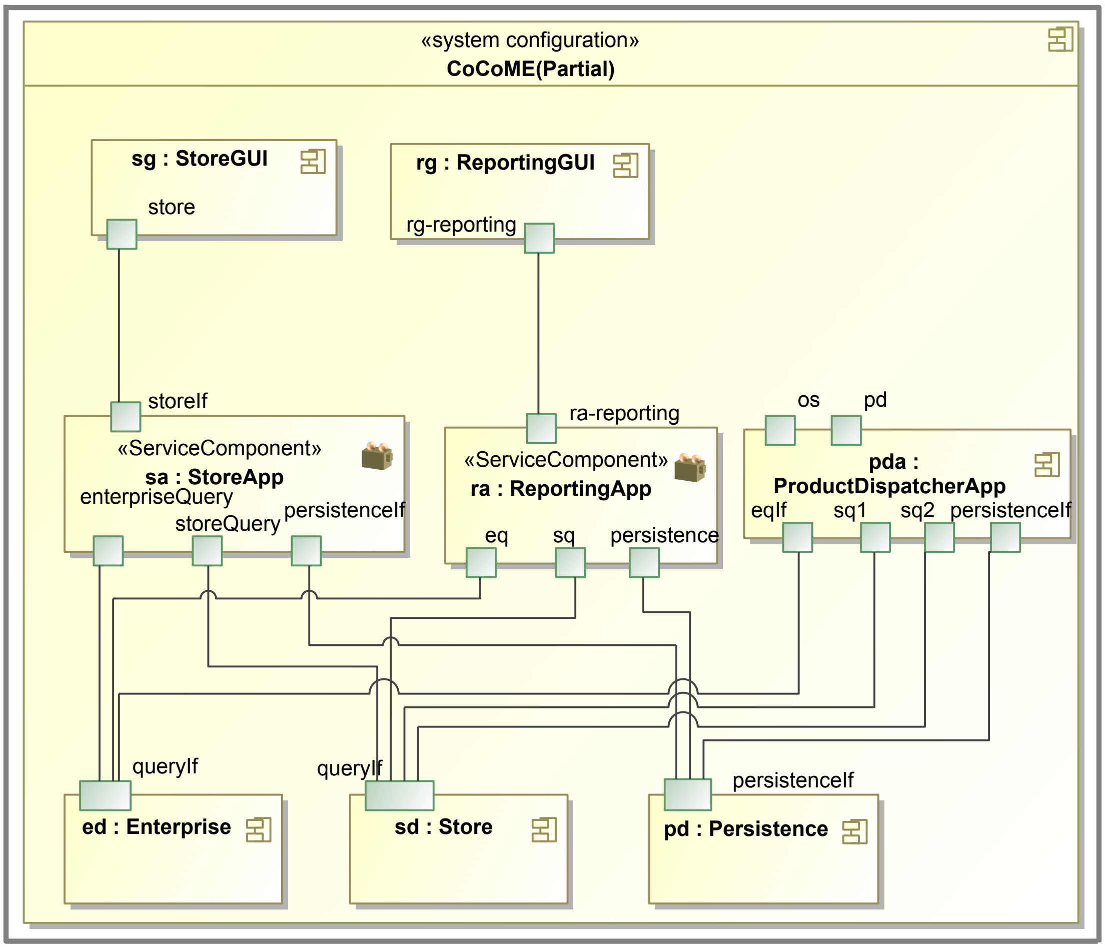


Figure 7.2: Overall system configuration of the Trading System, part one.

grained description of this part of the system configuration, the parts are more detailed; they are typed, define ports, and are connected by connectors.

The extensional statements that are generated out of the cut-out depicted in Fig. 7.2 according to the transformation definitions described in Sec. 6.3 claim the existence of a corresponding system configuration and the modelled parts and connectors. The following subset of the

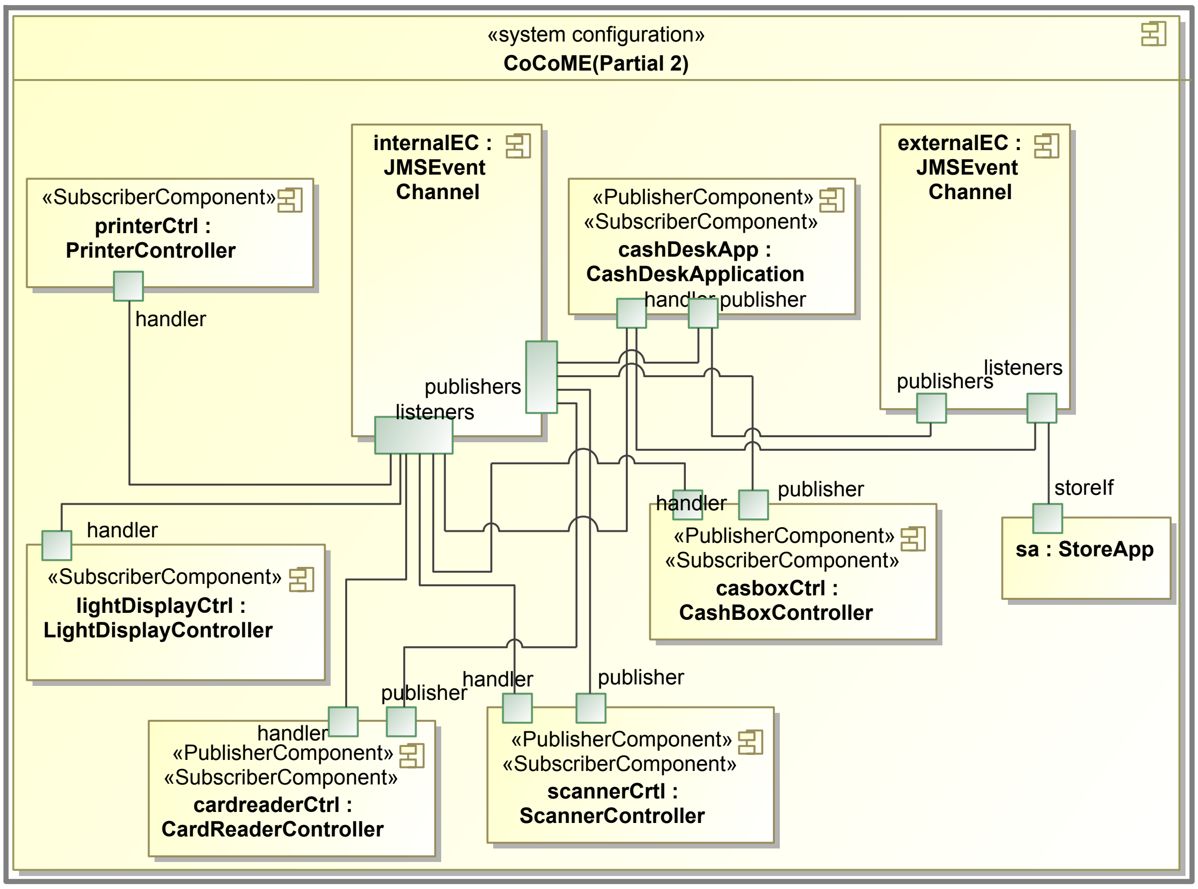

«system configuration»
CoCoME(Partial 2)
internalEC : JMSEvent Channel
publishers
listeners
externalEC : JMSEvent Channel
publishers
listeners
«SubscriberComponent» printerCtrl : PrinterController
handler
«PublisherComponent» «SubscriberComponent» cashDeskApp : CashDeskApplication
handler
publisher
handler
«SubscriberComponent» lightDisplayCtrl : LightDisplayController
handler
publisher
«PublisherComponent» «SubscriberComponent» casboxCtrl : CashBoxController
storeIf
sa : StoreApp
publisher
handler
handler
publisher
«PublisherComponent» «SubscriberComponent» cardreaderCtrl : CardReaderController
«PublisherComponent» «SubscriberComponent» scannerCrtl : ScannerController

Figure 7.3: Overall system configuration of the the Trading System, part two.

generated extensional statements describes the reporting part of the system configuration:

$$
\begin{aligned}
&\{\,SystemConfiguration(e_{CoCoME}),\\
&Part(e_{rg}), Part(e_{ra}),\\
&Connector(e_{con}),\\
&hasType(e_{rg}, e_{ReportingGUI}), hasType(e_{ra}, e_{ReportingApp}),\\
&configurationPart(e_{CoCoME}, e_{rg}), configurationPart(e_{CoCoME}, e_{ra}),\\
&configurationPart(e_{CoCoME}, e_{con}),\\
&connectorSource(e_{con}, e_{rg-reporting}), connectorTarget(e_{con}, e_{ra-reporting}),\\
&sourceContext(e_{con}, e_{rg}), targetContext(e_{con}, e_{ra}), \ldots\}
\end{aligned}
\tag{7.6}
$$

A complete listing of the generated statements would be too exhaustive to depict here; the application of the transformation definitions and the transfer to the omitted elements is straightforward.

Figure 7.3 depicts the system configuration part dealing with the cash desk system and its components. The parts are the same as modelled in the architectural model but are completed with types, ports, and connectors between pluggable parts and event channels. As illustrated in

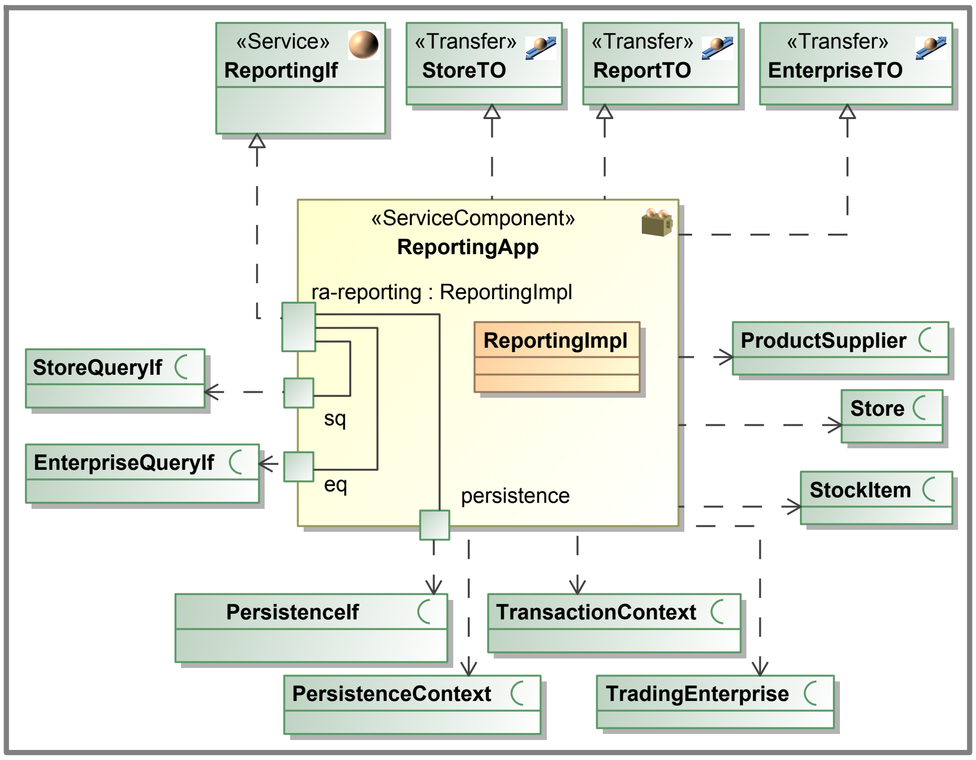


Figure 7.4: Cut-out of the design model: component ReportingApp.

App. C, all components used as types for publishers, subscriber, or event channels are properly stereotyped as PublisherComponent, SubscriberComponent, or EventChannelComponent. We will discuss their correct design in Sec. 7.2.

Figure 7.4 refines the type of the part ra from Fig. 7.2. There are several provided and required interfaces; some of them are stereotyped with design-specific stereotypes like Service for service interfaces or Transfer for classes whose instances are data transfer objects (see Sec. 6.2.2). Furthermore, it can be seen that the component is implemented by a single class called ReportingImpl. The only ports of the components are a provided port of type ReportingImpl and different required ports needed to access persistent data.

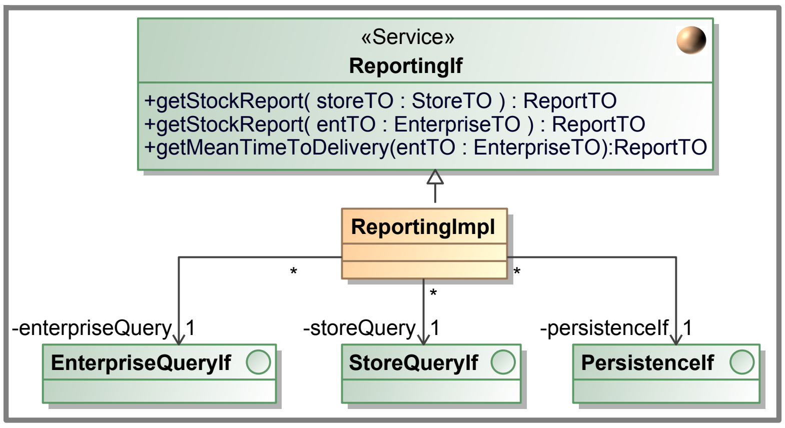

«Service»
ReportingIf
+getStockReport(storeTO : StoreTO) : ReportTO
+getStockReport(entTO : EnterpriseTO) : ReportTO
+getMeanTimeToDelivery(entTO : EnterpriseTO):ReportTO
ReportingImpl
*
*
*
-enterpriseQuery
1
-storeQuery
1
-persistenceIf
1
EnterpriseQueryIf
StoreQueryIf
PersistenceIf

Figure 7.5: Cut-out of the design model: implementation details of ReportingApp.

A subset of the extensional statements for this fragment of the design model is

$$\begin{aligned}\{ & \mathit{Component}(e_{ReportingApp}),\\ & \mathit{Interface}(e_{ReportingIf}), \mathit{Interface}(e_{ReportTO}), \mathit{Interface}(e_{StoreQueryIf}),\\ & \mathit{Service}(e_{ReportingIf}), \mathit{Transfer}(e_{ReportTO}),\\ & \mathit{providesInterface}(e_{ReportingApp}, e_{ReportingIf}),\\ & \mathit{providesInterface}(e_{ReportingApp}, e_{ReportTO}),\\ & \mathit{requiresInterface}(e_{ReportingApp}, e_{StoreQueryIf}),\\ & \mathit{Class}(e_{ReportingAppImpl}), \mathit{encapsulates}(e_{ReportingApp}, e_{ReportingAppImpl}),\\ & \mathit{implements}(e_{ReportingAppImpl}, e_{ReportingIf}),\\ & \mathit{ProvidedPort}(e_{ra-reporting}), \mathit{encapsulates}(e_{ReportingApp}, e_{ra-reporting}),\\ & \mathit{hasType}(e_{ra-reporting}, e_{ReportingAppImpl}), \ldots\}\end{aligned} \tag{7.7}$$

The details of the implementation of ReportingApp are depicted in Fig. 7.5. The three method declarations in the interface ReportingIf are transformed to *Signature* statements and corresponding *Parameter* statements. The three unidirectional associations are transformed into *Member* and *hasMember* statements that relate them to the class:

$$\begin{aligned}
&\{\ldots, Signature(e_{getStockReport-A}), Signature(e_{getStockReport-B}),\\
&\quad definesSignature(e_{ReportingIf}, e_{getStockReport-A}),\\
&\quad definesSignature(e_{ReportingIf}, e_{getStockReport-A}),\\
&\quad Parameter(e_{storeTO}), Parameter(e_{entTO}),\\
&\quad hasParameter(e_{getStockReport-A}, e_{storeTO}), hasParameter(e_{getStockReport-B}, e_{entTO}),\\
&\quad Member(e_{enterpriseQuery}), Member(e_{storeQuery}), Member(e_{persistenceIf}),\\
&\quad definesMember(e_{ReportingAppImpl}, e_{enterpriseQuery}),\\
&\quad definesMember(e_{ReportingAppImpl}, e_{storeQuery}),\\
&\quad definesMember(e_{ReportingAppImpl}, e_{persistenceIf}),\\
&\quad MethodBody(e_{getStockReport-A-impl}), MethodBody(e_{getStockReport-B-impl}),\\
&\quad definesMethod(e_{ReportingAppImpl}, e_{getStockReport-A-impl}),\\
&\quad definesMethod(e_{ReportingAppImpl}, e_{getStockReport-B-impl}),\\
&\quad hasSignature(e_{getStockReport-A-impl}, e_{getStockReport-A}),\\
&\quad hasSignature(e_{getStockReport-B-impl}, e_{getStockReport-B}), \ldots\}
\end{aligned} \tag{7.8}$$

Figure 7.6 depicts an example of a sequence diagram to describe the implementation of a method. The method getStockReport(StoreTO) returns a report about the amount of goods in the stock of a store given by the parameter . First, the method retrieves a reference to the persistence context to read from the database. Before it actually queries the database, it starts a transaction. After the database has been queried and the stock items are retrieved, a report is generated from the retrieved data, i.e. name of items, remaining amount, etc. The created report is returned.

According to the transformation rules defined in Sec. 6.3.2, the following set of extensional statements is generated. Again, the depicted set is not complete. Moreover, the constant symbols for the statement nodes are numbered because a naming scheme does not make sense in their case. However, note that in a UML model they constitute uniquely identifiable model elements and can hence be assigned a constant symbol. The following set contains mainly the statements generated for the synchronous call of *queryAllStockItems* at the store query

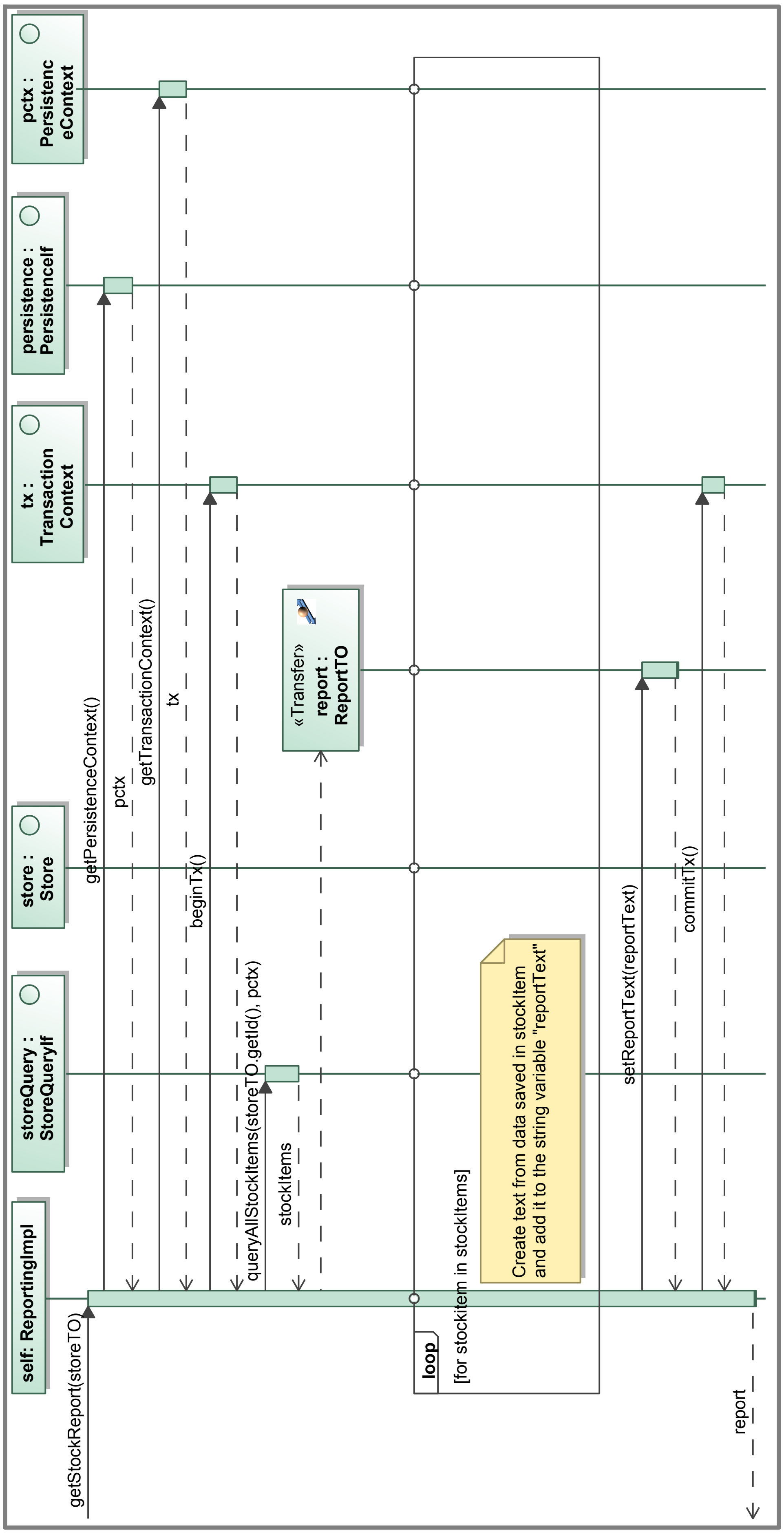

self: ReportingImpl
storeQuery : StoreQueryIf
store : Store
tx : Transaction Context
persistence : PersistenceIf
pctx : Persistenc eContext
«Transfer» report : ReportTO
getStockReport(storeTO)
getPersistenceContext()
pctx
getTransactionContext()
tx
beginTx()
queryAllStockItems(storeTO.getId(), pctx)
stockItems
loop
[for stockItem in stockItems]
Create text from data saved in stockItem and add it to the string variable "reportText"
setReportText(reportText)
commitTx()
report

Figure 7.6: Cut-out of the design model: sequence diagram for the partial behaviour of the implementation of ReportingApp.

interface:

$$\begin{aligned}
\{&\ldots, \mathit{LocalVariable}(e_{pctx}), \mathit{declaresVariable}(e_{getStockReport}, e_{pctx}),\\
&\mathit{LocalVariable}(e_{stockitems}), \mathit{declaresVariable}(e_{getStockReport}, e_{stockitems}),\\
&\mathit{SynchronousInvocation}(e_1),\\
&\mathit{containsStatement}(e_{getStockReport}, e_1),\\
&\mathit{invokedSignature}(e_1, e_{queryAllStockItems}),\\
&\mathit{invokedAt}(e_1, e_{storeQuery}),\\
&\mathit{ParameterBinding}(e_{pa1}), \mathit{ParameterBinding}(e_{pa2}),\\
&\mathit{parameterValue}(e_1, e_{pa1}), \mathit{parameterValue}(e_1, e_{pa2}),\\
&\mathit{formalParameter}(e_{pa1}, e_{parameter-id}), \mathit{actualParameter}(e_{pa1}, e_{storeid}),\\
&\mathit{formalParameter}(e_{pa2}, e_{parameter-pctx}), \mathit{actualParameter}(e_{pa2}, e_{pctx}),\\
&\mathit{returnedTo}(e_1, e_{stockitems}), \ldots\}
\end{aligned} \tag{7.9}$$

Intensional statements are generated for elements that are stereotyped with a stereotype from the design profile for UML described in Sec. 6.2 and Appendix B.2. Those model elements are, for example, service components, service interfaces, transfer objects, events, and event-processing related model elements as described in Sec. 6.2.3. In the following, a fragment for the design rules of the reference design model is given, whereas it is limited to some of the transfer object interfaces provided by the service component StoreApp, the service StoreIf, and the fact that StoreApp must also be valid subscriber component. The diagrams illustrating those elements can be found in Appendix C.

$$\begin{aligned}
\{&\mathit{onlyLegalMembers}(e_{\text{SupplierTO}}),\\
&\mathit{onlyGettersAndSetters}(e_{\text{SupplierTO}}),\\
&\mathit{onlyLegalMembers}(e_{\text{StockItemTO}}),\\
&\mathit{onlyGettersAndSetters}(e_{\text{StockItemTO}}),\\
&\mathit{hasServiceMethodsOnly}(e_{\text{StoreIf}}),\\
&\mathit{providesService}(e_{\text{StoreApp}}, e_{\text{StoreIf}}),\\
&\mathit{isValidSubscriberComp}(e_{\text{StoreApp}}), \ldots\}
\end{aligned} \tag{7.10}$$

As described in Sec. 6.2, these statements are intensional but local and, hence, are not architectural rules.

The depicted UML diagrams in this section are representative for similar diagrams that describe structure and behaviour of the CoCoME case study completely. Unfortunately, a complete description of the reference design model and the generated statements would go beyond the scope of this section. Appendix C provides the complete static reference design model and exemplary sequence diagrams. Together with the examples presented in this section, the appendix, and the transformation definition given in the Sections 6.2 and 6.3, the reader should be able to follow the results.

Relation symbol	Interpreting relation
SystemConfiguration	$\{e_{CoCoME}\}$
InnerPart	$\{e_{rg}, e_{ra}\}$
Connector	$\{e_{con}\}$
Component	$\{e_{ReportingApp}\}$
Interface	$\{e_{ReportingIf}, e_{ReportTO}\}$
Service	$\{e_{ReportingIf}\}$
Transfer	$\{e_{ReportTO}\}$
Class	$\{e_{ReportingAppImpl}\}$
ProvidedPort	$\{e_{ra-reporting}\}$
configurationPart	$\{(e_{CoCoME}, e_{ra}), (e_{CoCoME}, e_{rg})\}$
connectorSource	$\{(e_{con}, e_{rg-reporting})\}$
connectorTarget	$\{(e_{con}, e_{ra-reporting})\}$
sourceContext	$\{(e_{con}, e_{rg})\}$
targetContext	$\{(e_{con}, e_{ra})\}$
providesInterface	$\{(e_{ReportingApp}, e_{ReportingIf}), (e_{ReportingApp}, e_{ReportTO})\}$
requiresInterface	$\{(e_{ReportingApp}, e_{StoreQueryIf})\}$
encapsulates	$\{(e_{ReportingApp}, e_{ReportingAppImpl})\}$
implements	$\{(e_{ReportingAppImpl}, e_{ReportingIf})\}$
hasType	$\{(e_{ra-reporting}, e_{ReportingAppImpl})\}$

Table 7.2: Minimal system cut-out for reference design model.

A minimal structure for the extensional statements contained in the sets (7.6)-(7.9) is again straightforward: every relation symbol is interpreted by a relation that contains exactly those elements claimed by the extensional statements. Table 7.2 illustrates the relations of the minimal structure for the sets (7.6) and (7.7).

The minimal structure is also a minimal system, i.e. it fulfils the axioms from Φ_{CBSD}. Without depicting the complete minimal structure and the result of checking the validity of the axioms for it — in fact, the prototype that will be introduced in Chapter 8 can do this — they can be understood intuitively . The mapping between elements of the reference design model and the minimal structure is rather one-to-one: a modelled component C maps to an entity e_C that is in the relation interpreting the relation symbol *Component* in the minimal structure. This means that the axioms in Φ_{CBSD} that address multiplicities (see App. A) hold if the multiplicities in the model do no exceed those defined in Φ_{CBSD}. The remaining axioms, as introduced step-by-step in Sec. 5.3, are followed because the transformation definitions for UML models (see. Sec. 6.3) filter those models, whose generated set of statements would claim something which cannot be fulfilled by structures conforming to Φ_{CBSD}.

We also omit the proof that the minimal system for the set of extensional statements also satisfies the intensional statements of the reference design model. An informal explanation, why the intensional statements hold, will follow for the single architectural aspects that lead to corresponding design profile stereotypes in the next section.

7.2. Compliance Check of the Reference Design Model

As described in Sec. 5.5, the minimal systems of both, the architectural model and the reference design model, have to be merged in order to check whether the latter model is architecturally compliant. This is the case if and only if the merged system satisfies the architectural rules of the architectural model.

The following subsection will investigate whether this condition holds for the reference design model. Therefore, the results of checking the architectural rules will be elaborated step-by-step for each single architectural aspect.

7.2.1. Checking Layers

The rules for layers introduced in Sec. 6.2.1 refine dependencies between layers by stating how dependencies are created by relationships between components and interfaces. These relationships are compared with the existing “isAllowedToUse” relationships; if every dependency is inside a layer (or is directed to a system part not contained in a layer at all) or conforms to the allowed usage relations, everything is fine, and the system is correctly layered. This means for CoCoME that the set (8.3) of architectural rules has to be checked in the merged system.

In the merged system (see previous section), there are three layers mapped to three packages. Each of these packages, the three innermost packages in Fig. C.1, represents a complete layer; it holds that

$$mapsToPkg = \{(e_{GUILayer}, e_{GUI}), (e_{ApplicationLayer}, e_{Application}), (e_{DataLayer}, e_{Data})\}$$

Since, we would like to know the dependencies from a layer to another, we can analyse them package by package and take a look at the evaluation of the $dependsOn_i$ expressions. The following statements about the reference design model can be looked up in App. B.

At the GUI layer, there are only two components. $dependsOn_1$ is only true for them for interfaces from the application layer. $dependsOn_2$ can never be true for the GUI layer because there are no interfaces — the same holds for $dependsOn_3$ and $dependsOn_4$. $dependsOn_5$ can only be true for components used as parts in the system configuration because the interior of components contain only class-typed parts in all cases; there is no dependency in the system configuration that violates the allowed usage relations. $dependsOn_6(C_1, C_2)$ for a component C_1 from the GUI layer can only be true if C_2 is from the application layer because the only methods called from the GUI layer components are services at the application layer, or methods of transfer objects, which are defined in the application layer, too. For the same reason, the remaining $dependsOn_7$ and $dependsOn_8$ can neither be true for GUI components. In general, $dependsOn_8$ is only true for pairs of components of the inventory system, for which one of the other $dependsOn_i$ is already true; due to the fact, that in the inventory system each interface is provided by exactly one component, and the fact, that extension between interfaces is modelled only between interfaces of the same component, all the dependencies between components caused by synchronous invocation are captured at interface level.

Similar observations can be made for the application layer. All provided interfaces of components are defined in the application layer, all required interfaces are defined in the application layer or the data layer; $dependsOn_1$ is hence only true in "architecturally compliant" cases. Extension or usages between interfaces as members or signature parameters happen only between transfer object classes: they are all defined in the application layer, hence these expression becomes only true if both entities are "inside" the application layer; hence, $dependsOn_2$, $dependsOn_3$, and $dependsOn_4$ are true, if both or none of the arguments are from the application layer. Synchronous method calls are only made at target objects defined in the same layer or the data layer. Hence, $dependsOn_6$ and $dependsOn_7$ will only be true in architecturally compliant cases.

For the data layer holds, that all forms of dependencies are directed to elements inside the layer or elements not grouped in layers.

In conclusion, $depends_i$ holds only in architecturally compliant cases, i.e. for elements of the same layer, or for elements in layers for which an allowed usage relationships exists. Thus, the architectural rules regarding layers defined in Sec. 6.2.1 are true for the CoCoME system; the reference design model is architecturally compliant w.r.t the layers aspect.

7.2.2. Checking the Service-Oriented Appplication Layer Interface

To describe the architectural compliance checks for the service-oriented application layer, we first show that the design rules defined in the reference design model are fulfilled in the model itself, i.e. the model is intensionally complete.

The transfer objects interfaces refer only to primitive types[1] and other transfer object interfaces. Table 7.3 illustrates for every transfer object interface which other types are used. It is obvious that the constraints hold.

Figures C.8 and C.10 show the existing service interfaces in CoCoME — StoreIf and ReportingIf. It can easily be seen that both use only primitive types and transfer object interfaces as parameter and return types.

The exemplary sequence diagrams for the implementations of those services show that returned objects are always created in the same method. According to the design rules given in Sec. 6.2.2, the constraints for ServiceComponent are fulfilled, too.

Given this prerequisites, the architectural rules can be checked. The only $servicesExternallyUsed$ rule for the architectural model of CoCoME is defined for the application layer. The only layer in the system that is restricted to use only services of the application layer, is the GUI layer. Hence, the only possible connectors, which have to fulfil the constraints by the architectural rule are those connecting the following parts in the system configuration (see Fig. 7.2):

- Connector between part sg and part sa.
- Connector between part rg and part ra.

It is obvious that, given the fact that ReportingApp implements its service interface correctly, the second connector fulfils the constraint; its target port's type implements the service ReportingIf. The first connector, however, violates the constraint. Its target port storeIf does not

[1]Date is considered a primitive type.

Transfer object interface	**Types of member**
ComplexOrderTO	Date, long, ComplexOrderEntryTO
ComplexOrderEntryTO	long, ProductWithSupplierTO
EnterpriseTO	long, String
OrderTO	Date, long
OrderEntryTO	long
ProductTO	double, long, String
ProductAmountTO	long, ProductTO
ProductMovementTO	ProductAmountTO, StoreTO
ProductWithStockItemTO	double, long, String, StockItemTO
ProductWithSupplierTO	double, long, String, SupplierTO
ProductWithSupplierAndStockItemTO	double, long, String, StockItemTO, SupplierTO
ReportTO	String
SaleTO	Date, ProductWithStockItemTO
StockItemTO	double, long
StoreTO	long, String
StoreAndProductAmountTO	long, String, ProductAmountTO
StoreWithEnterpriseTO	long, String, EnterpriseTO
SupplierTO	long, String

Table 7.3: Interfaces stereotyped with Transfer and types of members.

only provide the service interface StoreIf but also CashdeskConnectorIf. This interface is *not* a service interface. Hence, $connectsToServiceOnly(e_{GUILayer}, e_{ApplicationLayer})$ is not true.

Note, that for the existing client in the GUI layer, i.e. the part sg, the object playing the role of storeIf does not appear as instance of CashDeskConnectorIf because this interface is not declared as required there. However, new GUI components could do so and invoke non-service methods. The simplest solution to this architectural mismatch would be to implement StoreIf and CashDeskConnectorIf separately and provide them by different ports.

7.2.3. Checking the Event-Driven Architecture

To show that the CoCoME cash desk system is architecturally compliant, we first have to show that the design model is intensionally complete, i.e. publishers, subscribers, and event channels are designed conforming to the design rules presented in Sec. 6.2.3.

As described in Sec. 4.1.4, the JMS framework has been used to realize the cash desk system. To apply the defined design rules in the reference design model, the relevant parts of that framework have to be modelled and stereotyped. Figure 7.7 depicts the important parts of the model regarding JMS.

The interface Message is annotated as event message interface[2]. As the rules state, it has to have a member which can contain those events to which it is connected by isCarrierFor

[2]More precisely, the interface javax.jms.ObjectMessage would have to be the stereotyped interface. For the

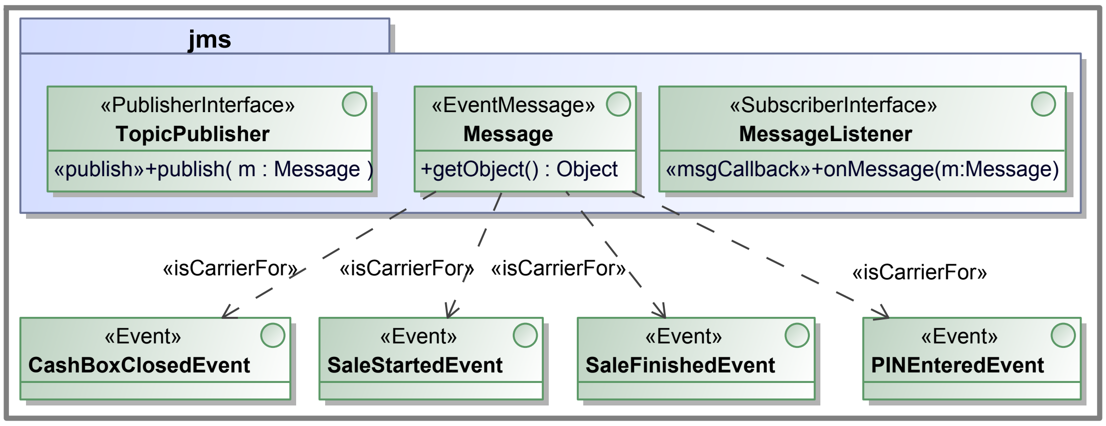


Figure 7.7: Design model of JMS entities annotated with design profile stereotypes.

dependencies. Such dependencies exist towards all defined events in the system, Fig. 7.7 shows four representatives. The member that can contain such an event is o of type Object which is the common base class of all classes, and which can hence carry also the defined events. Thus, the rule is valid for Message.

Furthermore, the interface TopicPublisher is stereotyped with PublisherInterface. Its publish signature fulfils the rules that have to hold for signatures marked by the publish stereotype; it has a parameter type marked by EventMessage, namely Message. The same holds for the interface MessageListener which fulfils the rules for subscriber interfaces. Concluding, the three depicted JMS interfaces are correct publisher interfaces, event message interfaces, and subscriber interfaces, respectively, in the sense of the design rules defined in the design profile.

As a next step, it can be shown that event channel components are correctly designed. Since there is no design model available for the JMS implementation used in CoCoME, a simple fictitious implementation is modelled as depicted in Fig. 7.8. Assuming, that an implementation of JMS is correct with regard to the specification, the behavioural model reflects the behaviour of that implementation; the call of the publish method causes the asynchronous forwarding of the message to all listening subscribers. For the reader familiar with the JMS details, please note, that this and the following models abstract from details like connections, sessions, etc. This simplest implementation depicted in Fig. 7.8 conforms to the rules generated for EventChannelComponent stereotyped components.

The design of components serving as types for publishers and subscribers is similar for all components in CoCOME playing one or more of these roles. Figure 7.9 depicts the structural design of the cash desk application component which is both, a subscriber and a publisher component. The structural models of other subscriber and publisher components of CoCoME are contained in App. C.

The cash desk application is a subscriber has hence to implement a SubscriberInterface and an EventHandlerInterface. Furthermore, for each of both interfaces it has to provide one port, respectively. This is given by the provided port handler and the interfaces it provides, ApplicationHandlerIf and MessageListener. The latter fulfils the relevant design rules as discussed

sake of simplicity, we annotate the superclass javax.jms.Message.

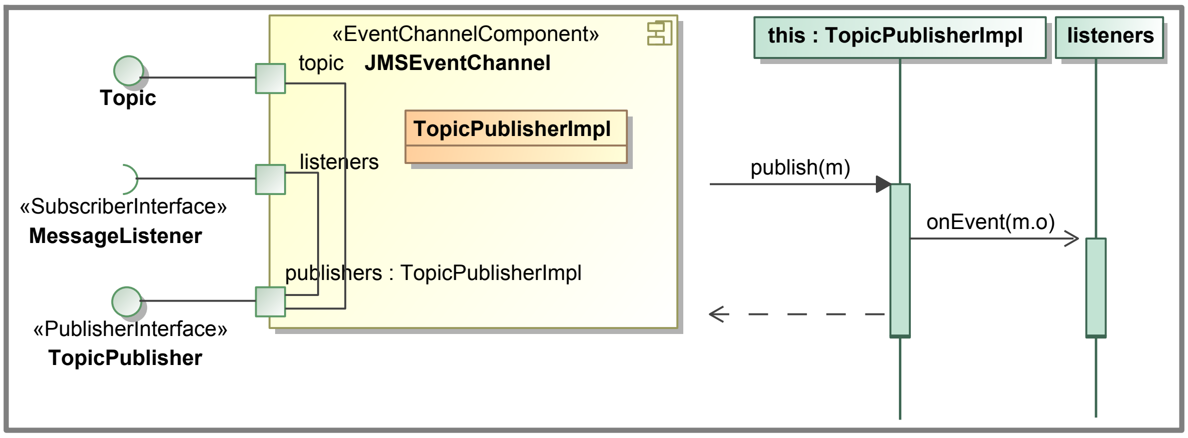


Figure 7.8: A fictitious implementation of an event channel in JMS.

above, and ApplicationHandlerIf is a correct event handler interface — all of its signatures are onEvent signatures, having a single Event parameter. Concluding, CashdeskApplication fulfils the rules regarding the structure of subscriber components.

As a publisher component, the cash desk application must specify a required port requiring an interface stereotyped with PublisherInterface. This is the case for intpublisher and extpublisher. Both require the interface TopicPublisher. Since a component instance will be plugged into two event channels at runtime, there are two corresponding required ports. So far, CashdeskApplication is also a conforming publisher component.

Figure 7.10 depicts the implementation of the onMessage method of the cash desk application, provided by the class CashdeskApplicationHandlerImpl. This method is implemented in other components in a similar way; the specific runtime type of the event, that is contained in the forwarded message, is determined by reflection. The different possible cases are considered in an if-statement. In each case, an onEvent method of the same object, which is also the event handler, is called. This is exactly the behaviour that is claimed by the corresponding rule in Sec. 6.2.3 for the special case, in which the class that implements the Subscriber interface is the same that implements the EventHandlerInterface. This means that — beside the structural aspects of subscribers — the behavioural rules regarding subscriber components are fulfilled, too.

The remaining components that are relevant for the cash desk system are designed the same way. They conform to the design rules, too, and are hence “valid” subscribers and publishers. Hence, the design model is intensionally complete with regard to the rules that are generated from this system part.

The architectural compliance can now be shown very easily. The only two event channels are internalEC and externalEC (see Fig. 7.3), that connect parts typed with properly designed subscriber and publisher components. As can be seen in Fig. 7.3, there is no connector between any of these parts. Hence, the architectural rules of statement (7.5) are fulfilled; they constitute a negation of an existential quantified statement over an empty set of entities. The overall formula is hence true, and architectural compliance is shown.

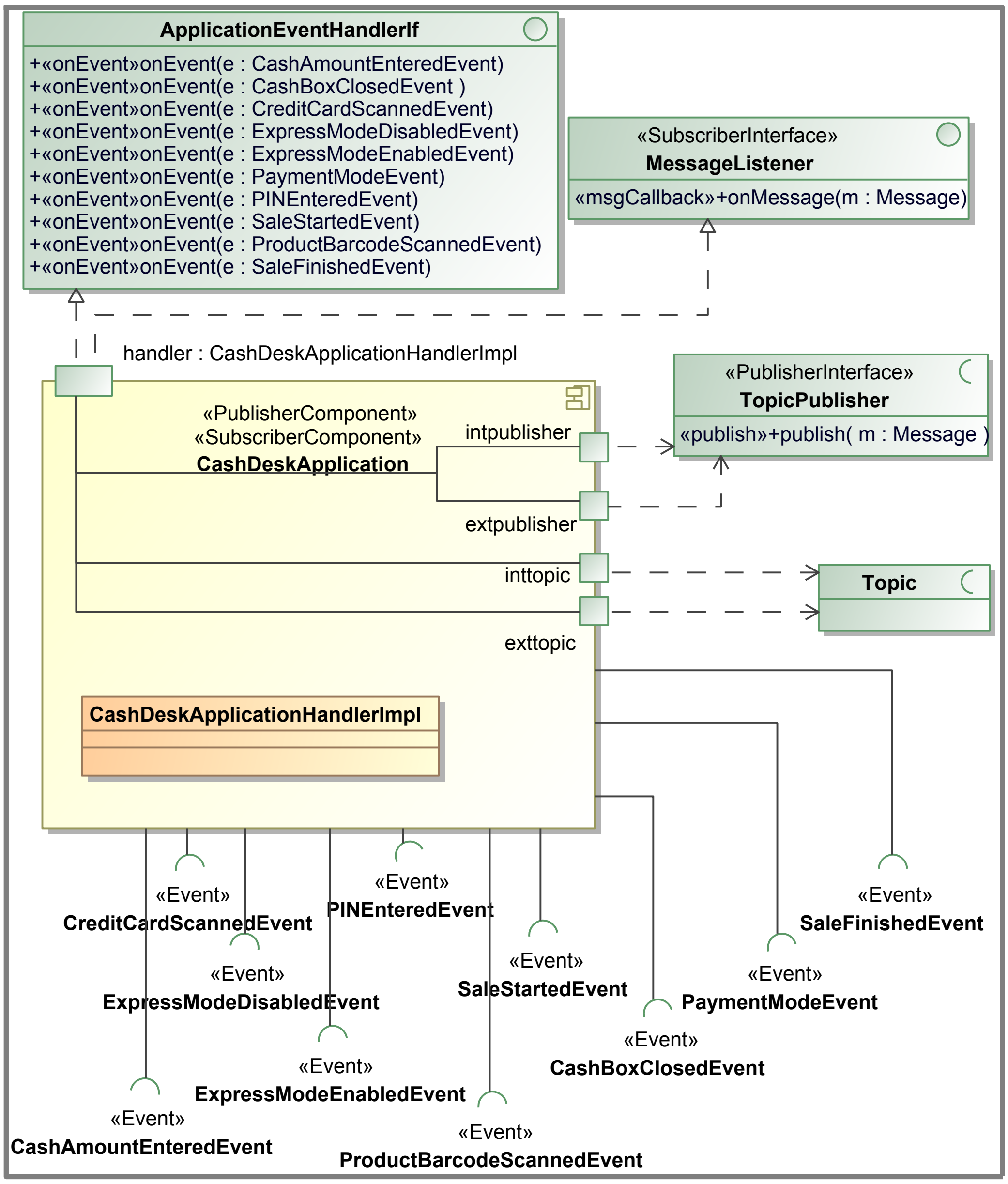

ApplicationEventHandlerIf
+«onEvent»onEvent(e : CashAmountEnteredEvent)
+«onEvent»onEvent(e : CashBoxClosedEvent)
+«onEvent»onEvent(e : CreditCardScannedEvent)
+«onEvent»onEvent(e : ExpressModeDisabledEvent)
+«onEvent»onEvent(e : ExpressModeEnabledEvent)
+«onEvent»onEvent(e : PaymentModeEvent)
+«onEvent»onEvent(e : PINEnteredEvent)
+«onEvent»onEvent(e : SaleStartedEvent)
+«onEvent»onEvent(e : ProductBarcodeScannedEvent)
+«onEvent»onEvent(e : SaleFinishedEvent)
«SubscriberInterface» MessageListener
«msgCallback»+onMessage(m : Message)
handler : CashDeskApplicationHandlerImpl
«PublisherInterface» TopicPublisher
«publish»+publish(m : Message)
«PublisherComponent» «SubscriberComponent» CashDeskApplication
intpublisher
extpublisher
inttopic
exttopic
Topic
CashDeskApplicationHandlerImpl
«Event» CreditCardScannedEvent
«Event» PINEnteredEvent
«Event» SaleFinishedEvent
«Event» ExpressModeDisabledEvent
«Event» SaleStartedEvent
«Event» PaymentModeEvent
«Event» CashBoxClosedEvent
«Event» ExpressModeEnabledEvent
«Event» CashAmountEnteredEvent
«Event» ProductBarcodeScannedEvent

Figure 7.9: Structure of the CashDeskApplication component.

7.3. Compliance Checks for the Modification Scenarios

The previous section has described the checking of the CoCoME reference design model. The results have revealed that there are no serious architectural mismatches in the design of the Trading System.

This section illustrates the proposed approach and the result of its application to the three modification scenarios introduced in Sec. 4.3. In each scenario, the reference design model, and the architectural model if necessary, are modified and violate one of the three architectural aspects of CoCoME as described in Sec. 4.2.

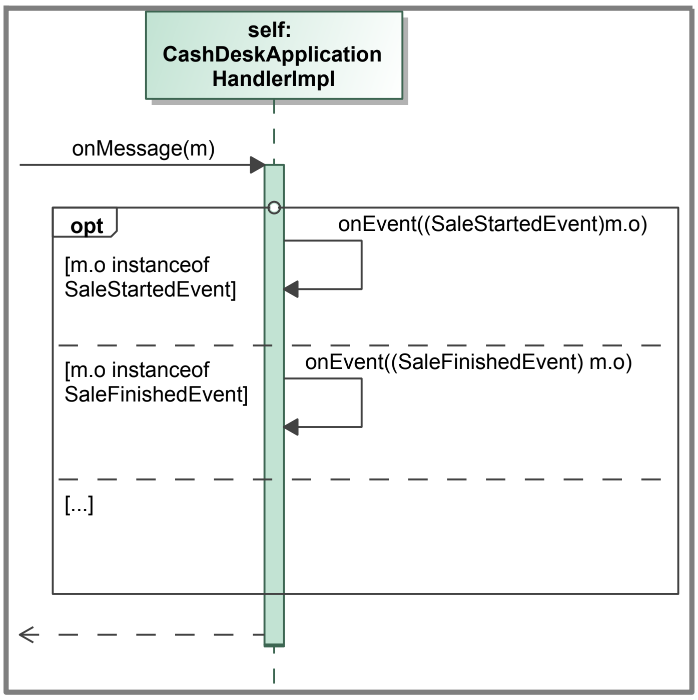

self:
CashDeskApplication
HandlerImpl
onMessage(m)
opt
[m.o instanceof SaleStartedEvent]
onEvent((SaleStartedEvent)m.o)
[m.o instanceof SaleFinishedEvent]
onEvent((SaleFinishedEvent) m.o)
[...]

Figure 7.10: Implementation of CashdeskApplication: how message calbbacks call event handlers.

7.3.1. Compliance in Modification Scenario 1

In this scenario, a logger component is added to the inventory system which can be used to log events, method calls, etc. during runtime. The component is integrated into the reference design model as depicted in Fig. 4.11 (p. 54). A new package containing new interfaces and a logger component is added. The CoCoME system configuration is complemented by a logger part as depicted in Fig. 7.11. Each component in the inventory system is going to log events, therefore they connect to the only provided port of the logger component. By this port, they can retrieve a log and write messages to it.

The architectural model is not modified for the initial solution to this scenario. The design model is contemplated by a new package, the two new additional interfaces, the logger component, new classes and method bodies for the implementation of the component. Hence, the set of generated extensional statements for the design models contains the following new

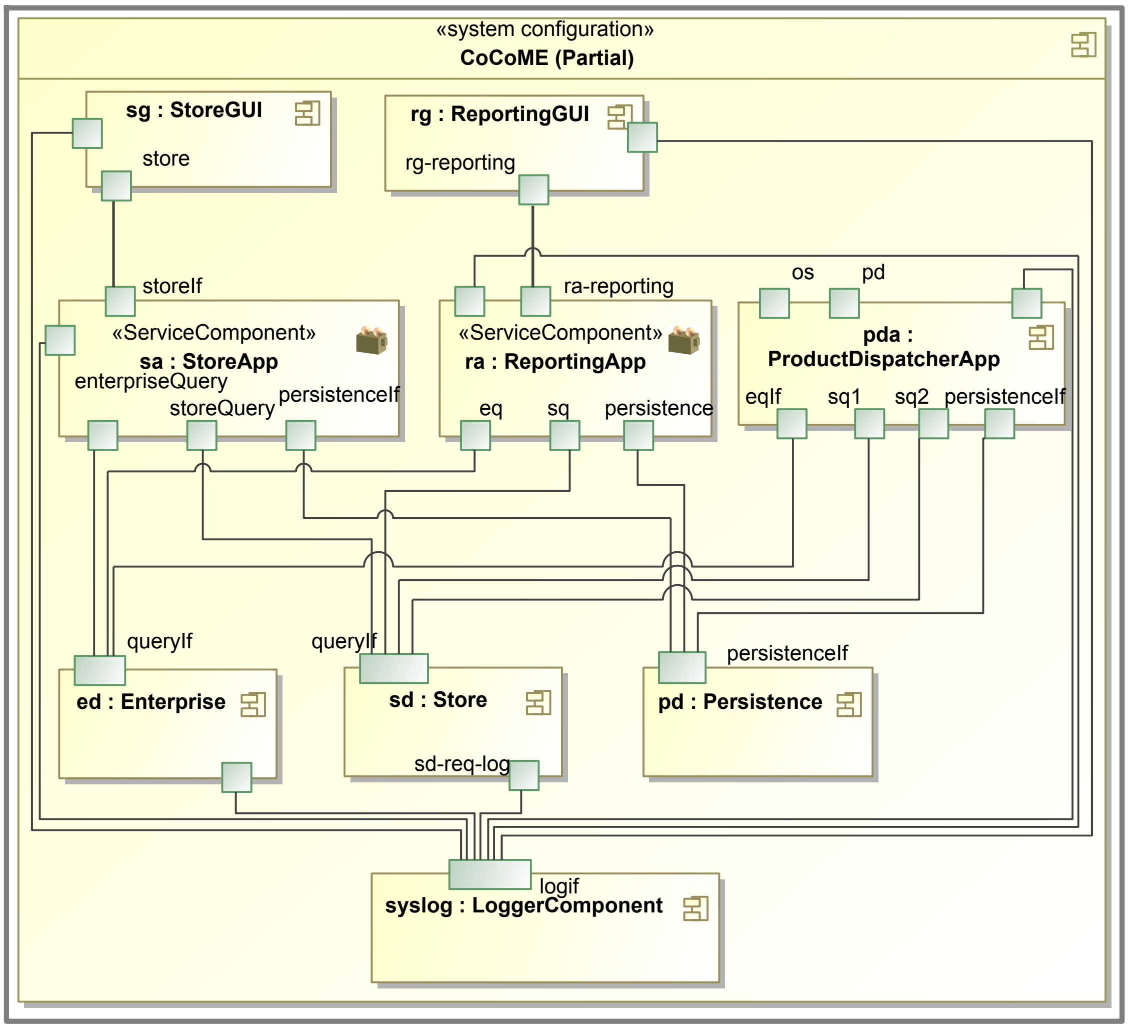


Figure 7.11: System Configuration with logging component.

statements[3]:

$$
\begin{aligned}
\{ & Package(e_{logging}), containsPkg(e_{Application}, e_{logging}), \\
& Component(e_{LoggerComponent}), Interface(e_{Logger}), Interface(e_{LogIf}), \\
& providesInterface(e_{LoggerComponent}, e_{Logger}), \\
& providesInterface(e_{LoggerComponent}, e_{LogIf}), \\
& ProvidedPort(e_{logif}), encapsulates(e_{LoggerComponent}, e_{logif}), \\
& providedInterface(e_{logif}, e_{LogIf}), \\
& Signature(e_{getLogger}), Signature(e_{log}), \\
& definesSignature(e_{LogIf}, e_{getLogger}), definesSignature(e_{Logger}, e_{log}), \ldots\}
\end{aligned}
\tag{7.11}
$$

Furthermore, the system configuration is changed because the newly added "logger part" is

[3]Statements regarding the implementation of the logger component are omitted since they are not relevant to the application scenario.

specified, as well as a number of connectors between already existing parts and the logger:

$$\begin{aligned}\{\, & Part(e_{syslog}), hasType(e_{syslog}),\\ & configurationPart(e_{CoCoME}, e_{syslog}),\\ & hasType(e_{syslog}, e_{LoggerComponent}),\\ & Connector(e_{sd-to-log}), configurationPart(e_{CoCoME}, e_{sd-to-log}),\\ & sourceContext(e_{sd-to-log}, e_{sd}), targetContext(e_{sd-to-log}, e_{syslog}), \ldots\}\end{aligned} \tag{7.12}$$

Moreover, several method calls are specified in existing method bodies that invoke the log method of the surrounding component's port connected to the logger part. Let us assume that the method queryLowStockItems (see Fig. C.23) is changed in the following way: before creating the first query it calls the method to retrieve a logger at the required port sd-req-log, which is that one connected to the logger: sd-req-log.getLogger(). For this statement, the following logical statements are created during the transformation:

$$\begin{aligned}\{\, & RequiredPort(e_{sd-req-log}), encapsulates(e_{Reporting}, e_{sd-req-log}),\\ & SynchronousInvocation(e_{call-getLogger}),\\ & containsStatement(e_{queryLowStockItems}, e_{call-getLogger}),\\ & invokedAt(e_{call-getLogger}, e_{sd-req-log}), \ldots\}\end{aligned} \tag{7.13}$$

Table 7.4 depicts the additional elements in the minimal system for the extensional statements of the modified design model according to the subsets (7.11)–(7.13)[4].

The evaluation of the architectural rules is now executed with the merged system consisting of the minimal system for the modified architectural model and the modified design model. The set of architectural rules is the same as for the unmodified CoCoME system in Sec. 7.2.

The architectural rule $\neg\, illegalDependenciesFromGroup(e_{DataLayer})$ evaluates to false in this scenario for the following reason. For each component in the data layer, there is a connector targeting at the logger component. As depicted in Tab. 7.4, at least $e_{sd-to-log} \in Connector$ is such an entity as the tuples

$$\begin{aligned}(e_{sd-to-log}, e_{sd-req-log}) &\in connectorSource\\ (e_{sd-to-log}, e_{logif}) &\in connectorTarget\\ (e_{LoggerComponent}, e_{logif}), (e_{Store}, e_{sd-req-log}) &\in encapsulates\\ (e_{sd-to-log}, e_{sd}) &\in sourceContext\\ (e_{sd-to-log}, e_{syslog}) &\in targetContext\end{aligned} \tag{7.14}$$

show. Additionally, as shown in the statement (7.13), there is also at least one synchronous call via this connector, or more precisely: a synchronous method call at the required port that is connected by the connector to the logger component[5]:

$$\begin{aligned}e_{call-getLogger} &\in SynchronousInvocation\\ (e_{queryLowStockItems}, e_{call-getLogger}) &\in containsStatement\\ (e_{call-getLogger}, e_{sd-req-log}) &\in invokedAt\end{aligned} \tag{7.15}$$

[4]The package structure is omitted for the sake of clarity.

[5]Not depicted but contained in the minimal system are the tuples modelling that there is a class in the Store component that provides an implementation for querLowStockItem.

Relation symbol	Interpreting relation
SystemConfiguration	$\{e_{CoCoME}\}$
Component	$\{e_{LoggerComponent}\}$
Interface	$\{e_{Logger}, e_{LogIf}\}$
Signature	$\{e_{getLogger}, e_{log}\}$
InnerPart	$\{e_{syslog}\}$
Connector	$\{e_{sd-to-log}\}$
ProvidedPort	$\{e_{logIf}\}$
RequiredPort	$\{e_{sd-req-log}\}$
SynchronousInvocation	$\{e_{call-getLogger}\}$
providesInterface	$\{(e_{LoggerComponent}, e_{Logger}), (e_{LoggerComponent}, e_{LogIf})\}$
definesSignature	$\{(e_{LogIf}, e_{getLogger}), (e_{Logger}, e_{log})\}$
configurationPart	$\{(e_{CoCoME}, e_{syslog}), (e_{CoCoME}, e_{sd-to-log})\}$
encapsulates	$\{(e_{LoggerComponent}, e_{logif}), (e_{Store}, e_{sd-req-log})\}$
sourceContext	$\{(e_{sd-to-log}, e_{sd})\}$
targetContext	$\{(e_{sd-to-log}, e_{syslog})\}$
connectorSource	$\{(e_{sd-to-log}, e_{sd-req-log})\}$
connectorTarget	$\{(e_{sd-to-log}, e_{logif})\}$
containsStatement	$\{(e_{queryLowStockItems}, e_{call-getLogger})\}$
invokedAt	$\{(e_{call-getLogger}, e_{sd-req-log})\}$

Table 7.4: Minimal system addition to model the logger components and calls to its interfaces.

Hence, the expression $dependsOn_6(e_{Store}, e_{LoggerComponent})$ evaluates to true (see Sec. 6.2.1):

$$\begin{aligned}
&dependsOn_6(e_{Store}, e_{LoggerComponent}) := Component(e_{Store}) \wedge \\
&\quad Component(e_{LoggerComponent}) \wedge \\
&\quad \exists con \exists p \exists q \exists r : InnerPart(p) \wedge InnerPart(q) \wedge Connector(con) \wedge hasType(p, e_{Store}) \wedge \\
&\quad (hasType(q, e_{LoggerComponent}) \vee \exists t : (Class(t) \wedge containsClass(e_{LoggerComponent}, t) \wedge \\
&\quad hasType(q, t)) \wedge \\
&\quad sourceContext(con, p) \wedge targetContext(con, q) \wedge connectorSource(con, r) \wedge \\
&\quad \exists c \exists m \exists s : (containsClass(e, c) \wedge hasMethod(c, m) \wedge \\
&\quad containsStatement(m, s) \wedge SynchronousInvocation(s) \wedge invokedAt(s, r))
\end{aligned} \tag{7.16}$$

for the variable binding

$$\begin{aligned}
con &= e_{sd-req-log} \\
p &= e_{sd} \\
q &= e_{syslog} \\
r &= e_{sd-req-log}
\end{aligned} \tag{7.17}$$

Since $inGroup(e_{Reporting}, e_{DataLayer})$ and $inGroup(e_{LoggerComponent}, e_{ApplicationLayer})$ evaluate to true, also $groupDependency(e_{DataLayer}, e_{ApplicationLayer})$ evaluates to true. This makes $illegalDependenciesFromGroup(e_{DataLayer})$ come true and, hence, violates the architectural rule mentioned above. This result is consistent to the scenario description of Sec. 4.3.1.

The solution proposed in the application scenario was to move the logger component to a utility layer that can be accessed from everywhere in the inventory system. Hence, there is an additional layer in the architectural model that is mapped to the package logging (see Fig. 4.12); the application layer, however, is no longer mapped to that package. Of course, the new layer defines a new architectural rule $\neg illegalDependenciesFromGroup(e_{UtilityLayer})$ that has to be checked, too.

Since, the logger does not use any of the other components, the new architectural rule holds. Furthermore, $inGroup(e_{LoggerComponent}, e_{ApplicationLayer})$ is not true anymore, such that $groupDependency(DataLayer, ApplicationLayer)$ becomes false, and the formerly violated architectural rule does hold now. Instead, $inGroup(e_{LoggerComponent}, e_{UtilityLayer})$ is true, which leads to an “allowed” group dependency only.

Hence, the compliance check returns the result that the design is now compliant with the modified architecture containing the additional utility layer.

7.3.2. Compliance in Modification Scenario 2

In this second modification scenario, the inventory system is extended by the possibility to determine orders that are planned to arrive on the current day. The scenario description given in Sec. 4.3.2 shows that the correct solution is developed in three steps after which an architectural compliance check should be executed successfully; after the first two solution attempts, checks should report errors.

The first solution step is depicted in Fig. 7.12. It corresponds to the leftmost section of Fig. 4.13 with the now introduced stereotypes. Moreover, it contains also the part of the system configuration relevant for this scenario.

In this phase of the solution in the second scenario, already the check, whether the design model is intensionally complete, will fail. As depicted, the signature for getSuppliersDeliveringToday uses the interface ProductSupplier as return type. This means that $(e_{getSuppliersDeliveringToday}, e_{ProductSupplier}) \in hasType$. Furthermore, it is also true that $e_{ProductSupplier} \notin Transfer$ and $e_{ProductSupplier} \notin PrimitiveType$, hence $isServiceSignature(e_{getSuppliersDeliveringToday})$ cannot be true (see statement (6.19)). This again causes $hasServiceMethodsOnly(e_{StoreIf})$ to be false – StoreIf is not a correct definition of a service interface. The compliance check delivers the expected result.

The second solution attempt addresses this issue. It changes the signature to return transfer objects. For this purpose, the designer defines a new transfer object interface named Supplier-WithOrderTO extending an existing one, as depicted in Fig. 7.13. In addition to the members it inherits, the interface refers to the interface ProductOrder from the persistent data model. Since ProductOrder is not stereotyped with Transfer, this is a wrong design of a transfer object interface.

The compliance check that is executed now detects the violation of a design rule. Due to the mentioned association, there is an entity m with $(e_{SupplierWithOrderTO}, e_m) \in hasMember$

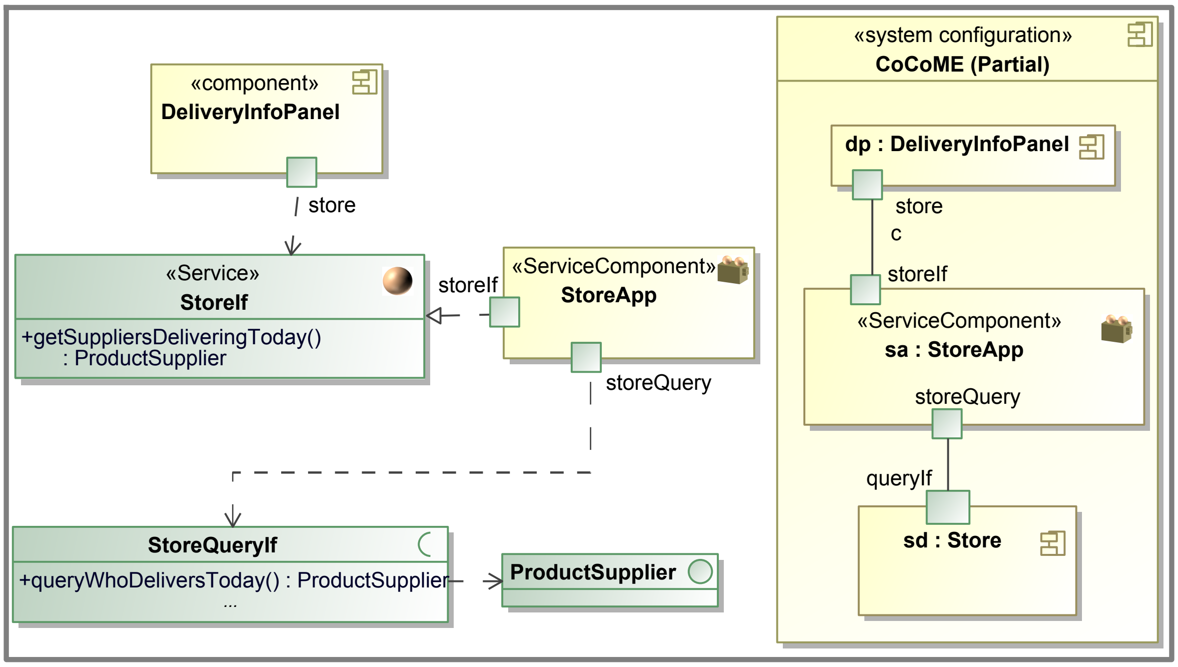


Figure 7.12: The first solution attempt to Modification Scenario 2.

and $(e_m, e_{ProductOrder}) \in hasType$. Furthermore, it holds that $e_{ProductOrder} \notin PrimitiveType$ and $e_{ProductOrder} \notin Transfer$. For this reason, $onlyLegalMembers(e_{SupplierWithOrderTO})$ does not hold, according to the statement (6.15). This means that SupplierWithOrderTO is an incorrect specification of a transfer object interface.

The final step towards a conforming design removes the association between the created transfer object interface and ProductOrder and replaces it by an association towards the existing transfer object interface OrderTO. Although not depicted, we assume that the implementation of getSuppliersDeliveringToday copies data from persistent objects into newly created transfer objects — in analogy to getProductsWithLowStock as depicted in Fig. 4.6. Hence, we can conclude that the design rules are fulfilled.

Now, the architectural rules have to be checked. The system configuration is the same as in step one depicted in Fig. 7.12. Furthermore, the architectural model is unchanged compared to the original one (see Fig. 7.1). This means that the set of architectural rules regarding the service-oriented application layer is still the same and consists only of the rule $onlyServicesExternallyUsed(e_{ApplicationLayer})$. If we assume, that the architectural mismatch regarding that rule in the reference model (see Sec. 7.2) was resolved as proposed, the only reason for a violation could be the connector c in Fig. 7.12. It connects two parts typed with components of layers that are connected by an *isAllowedToUseServices* dependency. The target port *storeIf* is providing StoreIf only, a correctly designed service interface. Hence, the providing component is a correctly designed service component, and the architectural rules hold. The architectural compliance check does not detect any errors after this third, final modification.

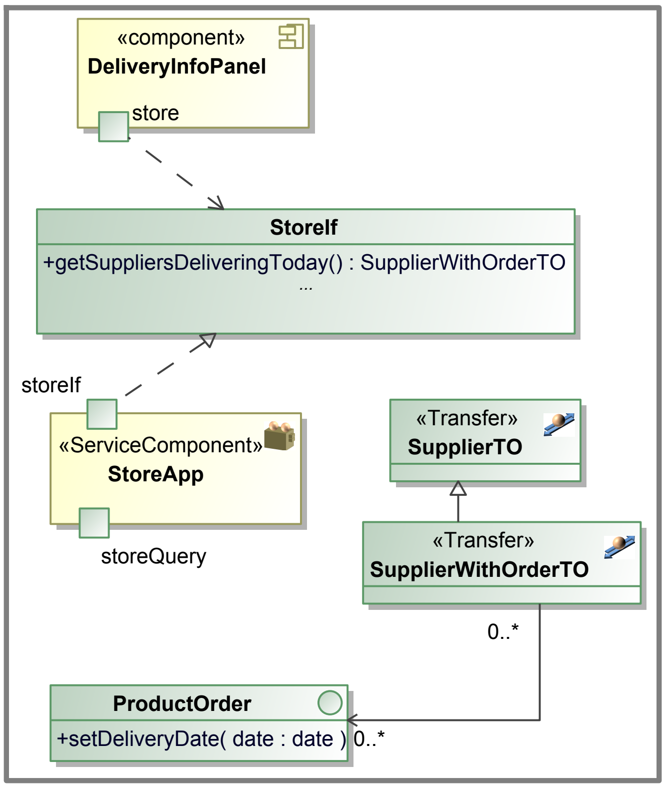


Figure 7.13: The second solution attempt to Modification Scenario 2.

7.3.3. Compliance in Modification Scenario 3

In this scenario, the cash box controller is modified to allow the input of a product bar code by the keyboard of the cash box. Since the components of the cash desk are very loosely coupled by the event-driven architecture, this additional functionality is easy to add; a BarcodeScannedEvent already exists and is published by the bar code controller if it scans a bar code successfully. The same event interface can now be published by the cash box controller when the cashier enters a bar code manually. All other components can react to this kind of event as usual and as already implemented.

However, the addition is not designed correctly. Figure 7.14 depicts the modified model and shows that the communication between cash box controller and cash desk application happens directly by bypassing the event channel. An architectural compliance check should detect this mistake.

Following the proposed approach, the design model containing the modified components will still be intensionally complete. There is only an additional required port for CashboxController, namely appHandler, requiring the ApplicationHandlerIf. The design rules refer only to the already existing elements of the component that are not changed. Without modelling the implementation of the component, we follow the informal description in Sec. 4.3.3 and assume that, when a bar code is entered, a corresponding event is only forwarded via apphandler.onEvent(), but not to the event channel.

The architectural model is the same as presented in Sec. 7.1.1 for the reference ver-

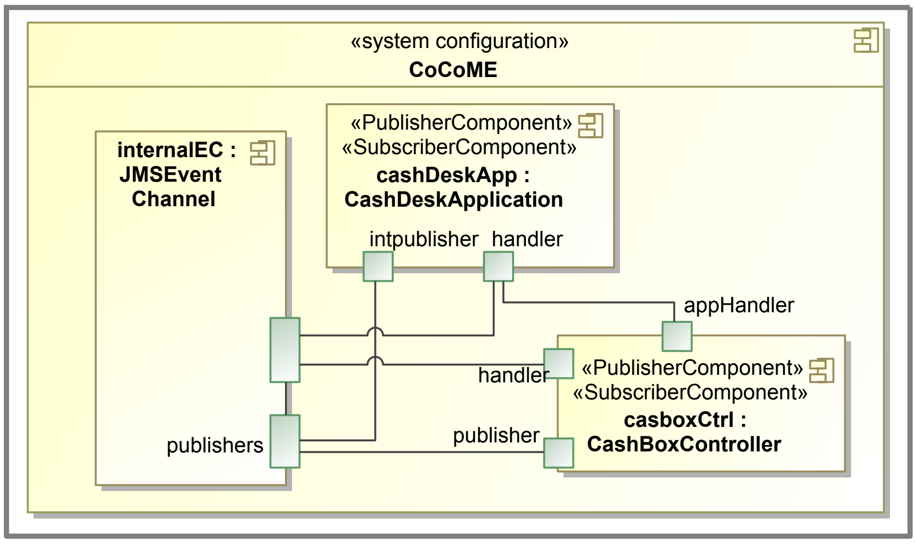

«system configuration»
CoCoME
internalEC : JMSEvent Channel
«PublisherComponent» «SubscriberComponent»
cashDeskApp : CashDeskApplication
intpublisher
handler
appHandler
handler
«PublisherComponent» «SubscriberComponent»
casboxCtrl : CashBoxController
publisher
publishers

Figure 7.14: Details of the incorrect solution to the third modification scenario.

sion of CoCoME. Hence, the architectural rules that have to be considered in this modification scenario are the same as before: *noBypassingOfChannel*(*internalEC*) and *noBypassingOfChannel*(*externalEC*) have to be checked.

While the latter statement is true as before — no component connected to the external event channel is changed — the first statement becomes false because of the connector between the required port appHandler, that is connected with the provided port handler of the cash desk application (see Fig. 7.14). Assuming the connector connecting both is referred to by e_c, the statement (6.44) evaluates to false for $x = e_{internalEC}$ and $c = e_c$:

$$\begin{aligned}
noBypassingOfChannel(e_{internalEC}) &:= \neg\,(\exists c \exists p \exists q : sourceContext(c,p) \wedge \\
&\quad targetContext(c,q) \wedge plugsInto(p, e_{internalEC}) \wedge plugsInto(q, e_{internalEC}) \wedge \\
&\quad \exists q' \exists i \exists j : (Event(j) \wedge connectorTarget(c,q') \wedge providedInterface(q',i) \\
&\quad \wedge dependsOn_4(i,j) \wedge isChannelFor(e_{internalEC}, j)))
\end{aligned}$$

p and q are the *cashboxCtrl* and *cashdeskApp* parts of the system configuration, and q' is the target port of the connector at *cashDeskApp*. The provided interface of this port, the event handler interface of the component, uses events distributed by the internal event channel. It holds, for example:

$$\{(e_{internalEC}, e_{ProductBarcodeScannedEvent})\} \subseteq isChannelFor$$
$$dependsOn_4(e_{ApplicationEventHandlerIf}, e_{ProductBarcodeScannedEvent}) = true$$

Thus, the inner existentially quantified part of *noBypassingOfChannel* evaluates to true. Since both addressed parts are plugged into the internal event channel (see Fig. 7.3), and therefore

$$\{(e_{cashboxCtrl}, e_{internalEC}), (e_{cashDeskApp}, e_{internalEC})\} \subseteq plugsInto$$

the overall formula evaluates to false; the architectural rule does not hold.

The solution to this issue is quite simple. The connector has to be removed, and the event has to be published to the internal event channel. The design model of this modification differs from the reference design model only in implementation details regarding the realization of the new functionality; design rules and architectural rules are not affected. We hence omit a proof that this solution is architecturally compliant.

7.4. Summary

In this chapter, the architectural compliance checks according to the proposed approach have been executed and analysed. To check the CoCoME case study, an architectural model, written in UML and using the developed architecture profile, has been developed. A reference design model has been created using UML and the design profile. The results of the transformations, for which the definitions have been given in Chapter 6, have been illustrated in excerpts, especially the architectural and design rules that result from applying the intensional component of the transformation definitions to the models of CoCoME.

The CoCoME system is architecturally compliant for the most part. The logical layer structure that separated the inventory system into a GUI, an application logic, and a data part, is not violated. The service-oriented interface that the application layer provides to the GUI layer is not consequently realized to the last detail. Because a port that is accessed from the GUI layer also provides a non-service interface, there is a small architectural mismatch. The event-driven architecture of the cash desk subsystem is realized correctly.

The architectural compliance checks for the three modification scenarios have returned the expected results. The layer assignment of a component that deals with cross-cutting functionality has been detected as a violation of the layer structure; contrarily, moving it to a separate utility layer that is allowed to be accessed from every other layer, has been detected as architecturally compliant solution to this issue. The checks in the second scenario, which has lead to the stepwise addition of a service, have detected architectural mismatches at several points; incorrect implementations of services and transfer object interfaces have been found. Finally, the bypassing of event channels in the third modification scenario has lead to negative results in the corresponding architectural compliance check as expected.

Chapter 8.

Design of a Logic-Based Compliance Checking Prototype

Contents

On two occasions I have been asked, “Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?” I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

Charles Babbage

This chapter introduces the prototypical realization of architectural compliance checking tool support basing upon the formal framework and concepts introduced in the chapters before. On the one hand, the chapter will present a specific tool prototype, called *Architecture Checker (ArCh)*, which implements architecture compliance checking for a dedicated set of supported meta models. The usage of this tool will be illustrated by applying it to a cut-out of the CoCoME modification scenarios (see Sec. 4.3). On the other hand, this chapter will describe the realization of the main concepts of the *formal* framework presented in Chapt. 5 as *implementation* framework; this framework realizes the formal representation of models and the checking of architectural rules according to the formal framework based upon the integration of a logical knowledge representation and reasoning system. This reusable framework can be used as backend for arbitrary compliance checking tools applying the concepts of this work. Thus, the framework allows the implementation of such tools with less effort.

The chapter is structured as follows. Section 8.1 will present ArCh as an example of a compliance checking tool and its usage. In Sec. 8.2, the overall architecture of a compliance checking tool like ArCh using the developed framework will be illustrated. Section 8.3 will describe the design of the framework, called *Architectural Compliance Checking Framework*, especially interfaces to extend and to use the backend. In Sec. 8.4, we will take a look upon the knowledge representation system PowerLoom used in the framework; Section 8.5 concludes this chapter.

8.1. The Compliance Checking Prototype from the User Perspective

The tool prototype *ArCh* realizes architectural compliance checking according to the proposed approach. ArCh is implemented as plugin [CR08] for the popular open-source integrated development environment Eclipse [ML05]. It integrates hence seamlessly with the look-and-feel and the functionality of any Eclipse installation it is plugged into.

In the following, we will see step-by-step how the software architect uses ArCh to execute the architectural compliance checks as required in the first modification scenario, the addition of a logger component to the basic CoCoME system (Sec. 4.3.1). We assume that Eclipse is used for the development of the CoCoME system. The architecture and design models are available as XMI files [Obj07]; XMI is a XML dialect to represent and exchange models between different tools. Many CASE tools, especially tools of the Eclipse Modelling Framework (EMF) [SBPM09], are able to import and export models as XMI files.

ArCh is automatically loaded as plugin when the containing Eclipse environment is started. The software architect invokes the functions of ArCh via the context menu of the project explorer, as depicted in Figure 8.1. The models, or in general the documents, that need to be checked are grouped by *configurations* indicating which set of models have to conform to which architectural model. ArCh is able to manage several configurations at the same time, allowing the architect to check architectural compliance in several projects. The architect can see its configurations by clicking “Manage Configurations”.

The ArCh Configuration Manager shows all available configurations (see Fig. 8.2). In the example, the architect has already created a configuration which contains two documents, listed in the right area of the dialogue as document entries. The document named “Reference Layers Arch” contains a model of the architectural layers of CoCoME as depicted in Fig. 4.11 (p. 54). The model named “Design Model (Initial Logger Solution)” contains the design model which includes the incorrect embedding of the logger component into the application layer.

Each document entry lists furthermore the physical resource representing the model and, below the heading “Type”, its meta model. Note, that the architectural model is not a UML model as assumed in previous chapters. The depicted meta model is equivalent to the cut-out

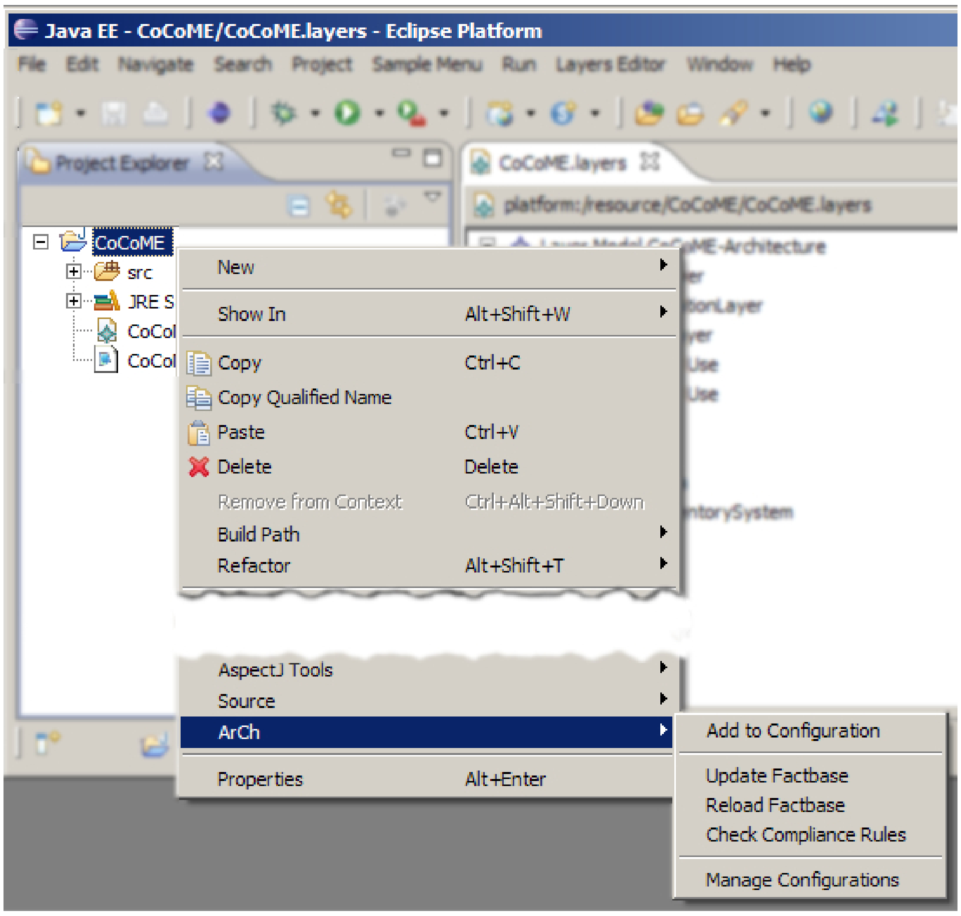
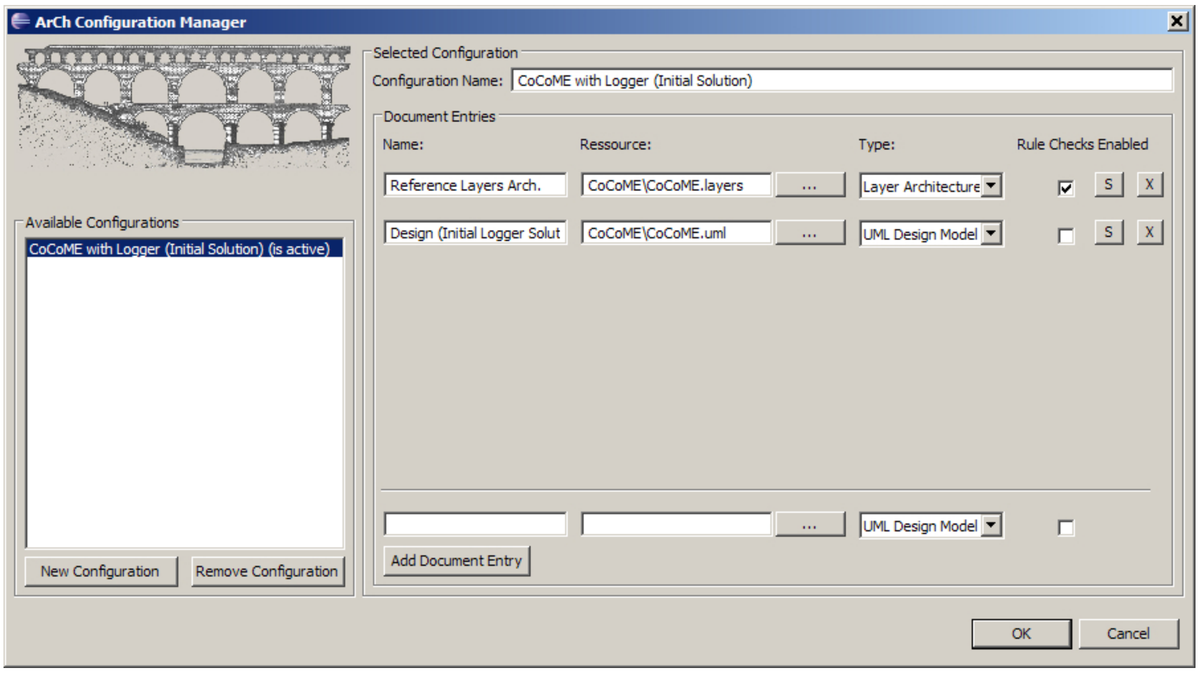


Figure 8.1: The ArCh context menu.

of the architecture profile for UML dealing with layers and used instead to demonstrate the integration of different meta models into the compliance checking tool. Together with each meta model, the transformation definition according to Sec. 5.5 is made available to the tool, such that the architectural rules of a model as instance of a meta model can be determined.

Moreover, for each single document entry, it can be activated whether the contained rules — i.e., the intensional statements of a model — should be checked. Since the transformation definitions for standard UML do not define any intensional statements (see. Sec. 6.3), enabling the checkbox for the design model in the example would not have any effect. Each entry can be saved in case of changes (e.g., modified name or path to resource) or removed from the configuration (buttons “S” and “X”). The architect reassures that the entries refer to the correct models and that the selected configuration is the “active one”; this is indicated in the list of available configurations at the left hand side of the configuration manager. The active configuration is that one being loaded into the knowledge representation system for the next compliance checks.

After closing the configuration manager, the actual compliance check is executed again via the context menu of the project explorer and the item “Check Compliance Rules”. In the background, the models contained in the active configuration are loaded; minimal systems



Available Configurations

CoCoME with Logger (Initial Solution) (is active)

Selected Configuration

Configuration Name: CoCoME with Logger (Initial Solution)

Document Entries

Name:	Ressource:	Type:	Rule Checks Enabled
Reference Layers Arch.	CoCoME\CoCoME.layers	Layer Architecture	☑
Design (Initial Logger Solut	CoCoME\CoCoME.uml	UML Design Model	☐
		UML Design Model	☐

Figure 8.2: The ArCh Configuration Manager.

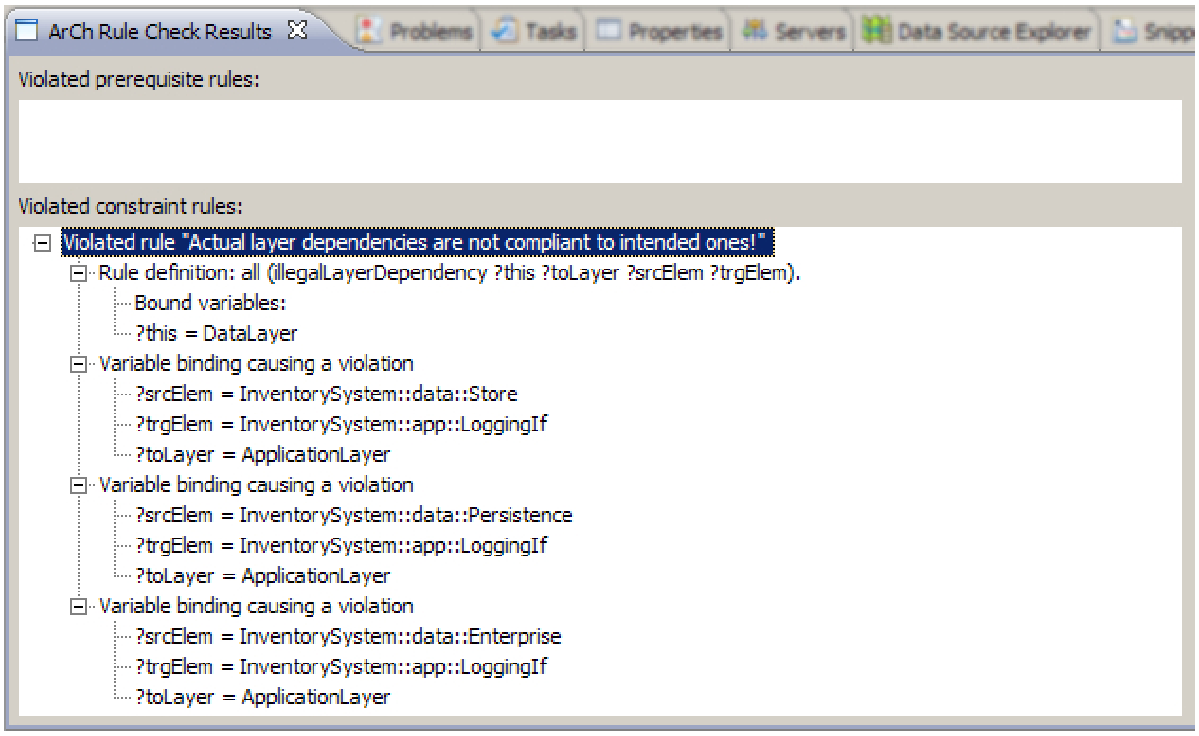
of the models are generated for the single models and merged into a common knowledge base. After that, the rules of all documents[1], especially the architectural rules defined by the architecture model, are executed as queries over the knowledge base. In a separate view, the "ArCh Rule Check Result" view, the software architect can inspect violated rules. Figure 8.3 shows the rule check result for the initial solution to the first modification scenario.

The view is separated into two different lists for different kind of rules which somehow prioritize the execution of rules. *Prerequisite* rules are utility rules that ensure that preconditions are met for the execution of constraint rules. Constraint rules are executed only if all prerequisite rules hold. This way, a very basic prioritization of rules is implemented. Prerequisite rules normally catch cases in which constraint rules cannot be checked in a meaningful way. For example, a prerequisite rule for the checking of layers ensures that every layer is mapped to a package; otherwise, checking illegal layer dependencies will not detect any violations at all.

Constraint rules are used (in this example) to express the actual architectural rules. As the result view shows in Fig. 8.3, they are formulated as the negation of the actual architectural rule. The corresponding query returns hence those elements that cause rule violations. The marked line shows a human-readable text describing the violated rule, in this case a layer depending on something that it is not allowed to depend on. In the next line, the formal definition of the violated rule (see Sec. 8.3.2) follows; it furthermore shows the binding of the variable `?this`) (see also Sec. 5.5.1) indicating for which layer the architectural rule is violated. This is followed by all variable bindings causing the query to be true and the rule hence to be violated.

It is visible that the result corresponds to the expectations for the initial solution to Modification Scenario 1 (see Sec. 4.3.1). Each component of the data layer using the logging interface,

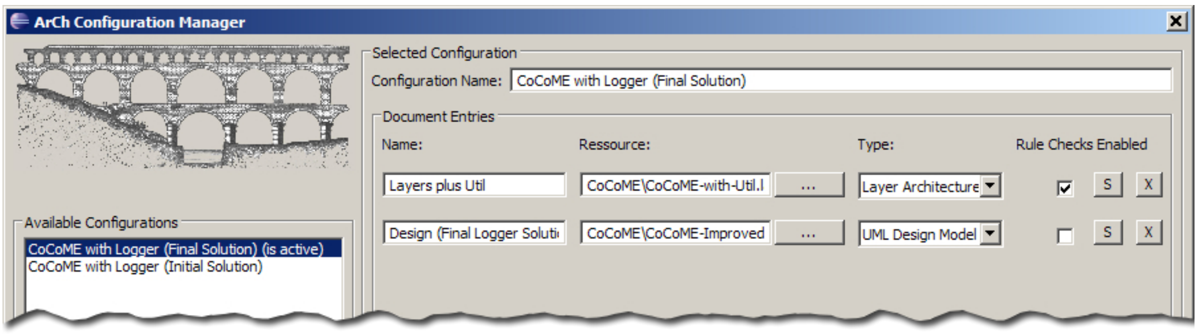
[1]Unless a document has been marked to be ignored for checking its rules, see Fig. 8.2.



```
Violated prerequisite rules:

Violated constraint rules:
⊟ Violated rule "Actual layer dependencies are not compliant to intended ones!"
   ⊟ Rule definition: all (illegalLayerDependency ?this ?toLayer ?srcElem ?trgElem).
        Bound variables:
        ?this = DataLayer
   ⊟ Variable binding causing a violation
        ?srcElem = InventorySystem::data::Store
        ?trgElem = InventorySystem::app::LoggingIf
        ?toLayer = ApplicationLayer
   ⊟ Variable binding causing a violation
        ?srcElem = InventorySystem::data::Persistence
        ?trgElem = InventorySystem::app::LoggingIf
        ?toLayer = ApplicationLayer
   ⊟ Variable binding causing a violation
        ?srcElem = InventorySystem::data::Enterprise
        ?trgElem = InventorySystem::app::LoggingIf
        ?toLayer = ApplicationLayer
```

Figure 8.3: ArCh checking results for initial solution to modification scenario one.



Available Configurations
- CoCoME with Logger (Final Solution) (is active)
- CoCoME with Logger (Initial Solution)

Selected Configuration

Configuration Name: CoCoME with Logger (Final Solution)

Document Entries

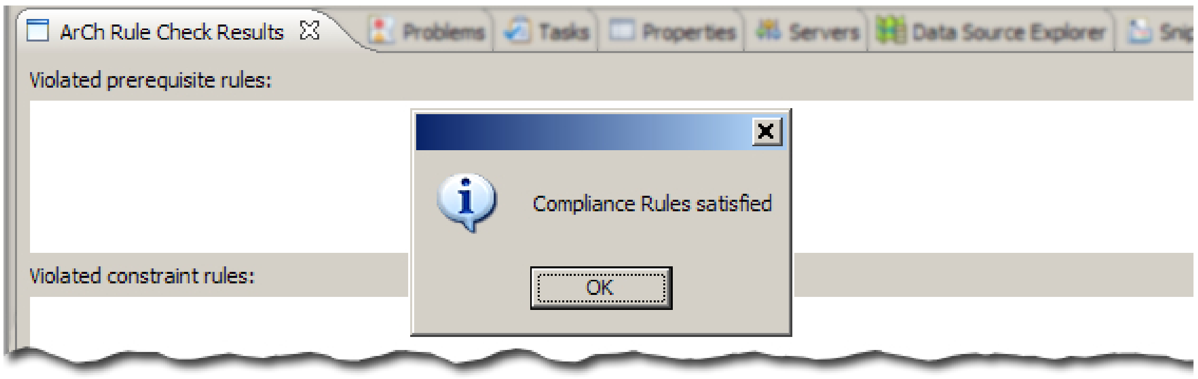
Name:	Ressource:	Type:	Rule Checks Enabled
Layers plus Util	CoCoME\CoCoME-with-Util.l	Layer Architecture	✓
Design (Final Logger Soluti	CoCoME\CoCoME-Improved	UML Design Model	☐

Figure 8.4: Adding a configuration for the final solution for modification scenario one.

which are Store, Enterprise, and Persistence, causes a rule violation.

The architect creates a second configuration (Fig. 8.4) that corresponds to the final and correct solution of the first modification scenario — moving the logger component and its interfaces to a cross-cutting utility layer that can be accessed from every layer. The new configuration contains the modified architecture model and the design model describing the correct solution as depicted in Fig. 4.12 (p. 55).

A new compliance check for this configuration, which is activated before, returns without error, as Fig. 8.5 shows.



Violated prerequisite rules:

Compliance Rules satisfied

Violated constraint rules:

Figure 8.5: The architectural rules hold for the final solution in Modification Scenario 1.

8.2. Architecture of a Compliance Checking Tool

This section presents the logical architecture of ArCh and the embedding of ArCh into Eclipse. The described structures can be understood as blue-prints for the architecture of compliance checking tools following the formal framework proposed in this thesis.

8.2.1. Logical Overview

The logical structure of ArCh can be separated into two main building blocks, the frontend and the backend. This top-level architecture is depicted in Fig. 8.6.

The frontend of ArCh provides the user interface and the connection to the backend according to the realized workflow; it ensures, for example, that the compliance check functionality of the backend is executed when the corresponding item of the ArCh context menu has been selected, and presents the checking results afterwards (see Fig. 8.1 and Fig. 8.3).

While the frontend of an architectural compliance checking tool is tool-specific in large parts, the main building block of the backend is a reusable tool-independent architecture compliance checking framework (ACC framework, see Fig. 8.6). Its functionality can be invoked by an application interface module which allows frontends to create and manage configurations, execute compliance checks, etc. The framework realizes the core aspects of the formal framework introduces in Chapt. 5, i.e. checking architectural compliance between models based on logical architectural rules that are evaluated on τ_{CBSD}-systems. For this purpose, it applies the query mechanisms of a knowledge representation system. The overall implementation of the framework is independent of any specific meta model in the sense that its functionality is realized without referring to concrete types of models that participate in the process of architecture compliance checking. The framework defines instead an interface for *document wrappers*. A document wrapper implementing this interface enables the framework to retrieve the minimal systems of models of one specific meta model. The backend of ArCh, for example, provides wrappers for standard UML and for the architectural meta model for layers used in the previous section. Each wrapper for itself, however, is reusable in any compliance checking tool using the ACC framework. Together, the actual set of wrappers and the ACC framework make up the overall backend of an ACC tool.

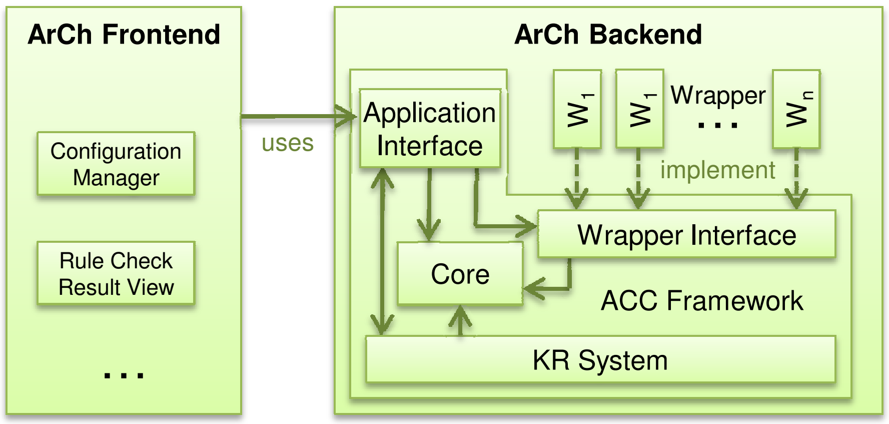


Figure 8.6: Logical top-level structure of architectural compliance checking (ACC) tools applying the developed framework.

The details of the backend will be subject of Sec. 8.3.

8.2.2. Eclipse Integration

The Eclipse platform is not a monolithic system. It consists of a relatively small core or kernel to which additional plugins can be loaded at start-up. In fact, the kernel of Eclipse contains an implementation of the OSGi specification [OSG03] which describes a component model (see Sec 2.3.2); Eclipse plugins can be seen as components conforming to this component model. The kernel is hence able to load such plugins and provides the infrastructure to connect plugins and to enable them to interact. The minimal set of plugins required to add custom-made applications as plugins is called the *Eclipse Application Framework (EAF)*.

A plugin itself is a program written in Java implementing a certain interface and specifying additionally the dependencies towards other plugins and provided services. This information is declared in configuration files and is used by the Eclipse infrastructure to ensure that plugins can find each other and connect to use each other's functionality if required. For more details on Eclipse plugins, please refer, for example, to [CR08].

The ArCh prototype is realized as a set of several Eclipse plugins. The plugin structure of ArCh is depicted in Fig. 8.7. The frontend of ArCh is realized as single plugin. It defines, as mentioned above, the graphical user interface and declares additional configuration information which allows the Eclipse infrastructure to integrate the plugin properly into the standard Eclipse framework. For example, it is specified that the ArCh menu is integrated into the context menu of the project explorer.

In contrast to the logical frontend, the backend is distributed over different plugins. Each wrapper is encapsulated by a single plugin as well as the ACC framework. Wrapper plugins require an instance of the framework plugin to register the encapsulated wrappers as available document wrappers. The set of available wrappers can be retrieved from the framework by arbitrary frontends. This way, an existing compliance checking tool can be simply adapted to new meta models by loading new wrapper plugins into Eclipse. At the moment, another wrapper plugin for Java is developed, using the built-in facilities of Eclipse to traverse and wrap

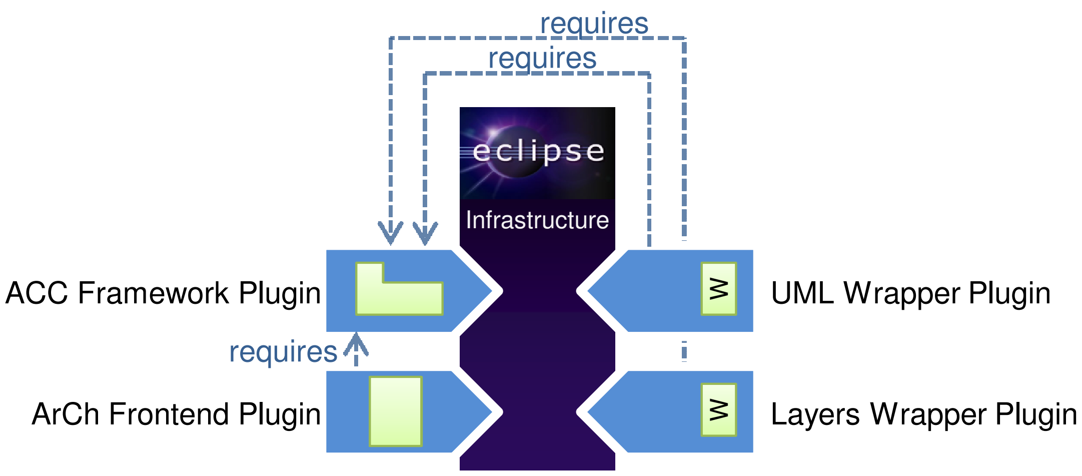


Figure 8.7: ArCh integration into Eclipse.

Java source code.

Furthermore, the ACC framework is encapsulated by a single plugin. The configuration coming with the *ACC Framework Plugin* ensures that everything that is required by wrapper plugins and frontends, i.e. the core functionality of compliance checking, all relevant data types, and the wrapper interface, is visible to the environment.

The dependency relations depicted in Fig. 8.7 refer to the static dependencies of the overall ArCh system. The required calls from the frontend to functions provided by the framework have to be specified in the source code of the frontend. Furthermore, each wrapper needs to implement an interface defined in the framework and calls operations for registering itself at the framework. Nevertheless, there are dynamic dependencies between the framework and the wrappers. According to the common “don’t call us, we call you” principle of frameworks, the ACC framework will invoke wrapper methods. Statically, these calls are hidden by specifying calls in the framework’s source code that are invoked at abstract interfaces.

8.3. The Architecture Compliance Checking Framework

The ACC framework provides reusable functionalities for architecture compliance checking tools. It implements the main functions of compliance checking independently of the specific meta models whose instances need to be checked by a specific compliance checking tool. The framework is implemented in Java and consists of four subsystems as depicted in Fig. 8.8. These subsystems are:

Core. The core subsystem provides the basic data structures to represent τ-structure as introduced in Sec. 5.2 in general, and implements basic operations to construct and to modify them. Moreover, it defines interfaces to traverse such structures, for example in order to realize code generators transforming those structures into code for a logical knowledge representation system. Last but not least, the framework’s core provides data structures for the specification of logical statements in general, for example to represent architectural rules.

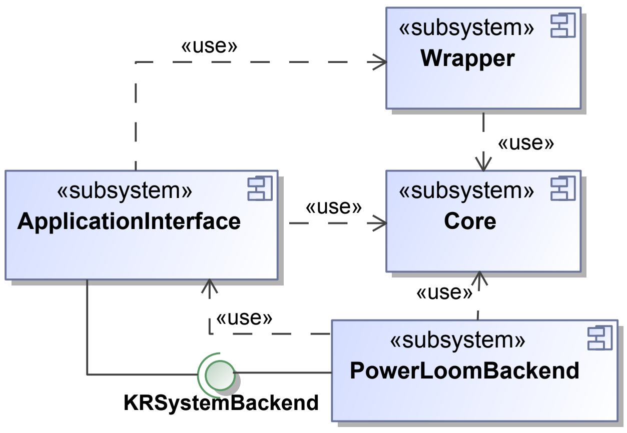


Figure 8.8: The subsystems of the ACC framework.

Wrapper. The wrapper subsystem consists of interfaces and classes for the purpose of integrating meta model-specific wrappers into instances of the framework such that instances of those meta models can be checked by tools using the framework. The main elements of this subsystem are functional modules and data types for registering and accessing wrappers at run-time, and a wrapper interface which must be implemented by the specific wrappers. These wrappers deliver a τ_{CBSD}-structure representing a minimal structure for the encapsulated model as well as the set of intensional rules that the model defines. For this purpose, the subsystem relies on the implementation of logical structures provided by the Core subsystem.

PowerLoomBackend The task of this subsystem is the integration of the PowerLoom knowledge representation system. It translates object structures, that conform to the data types defined in the Core subsystem, into PowerLoom knowledge bases and implements rules checking as execution of logical queries. It implements the interface defined in Core to traverse logical structures to realize a code generator for the representation of logical structures in PowerLoom. The subsystem implements furthermore an abstract interface KRSystemBackend, on which the ApplicationInterface.

ApplicationInterface. This subsystem represents the main interface for architecture compliance checking tools to configure the framework for the specific purpose, to manage the set of models participating in the compliance checking process, and to invoke the checking functionality. For this purpose, it defines functional interfaces for the access by frontends but also datatypes that are in addition used by the PowerLoomBackend. The realization of checking functionality follows roughly a common scheme: the data contained in a model is retrieved from the wrappers as logical structure and forwarded to the PowerLoomBackend which transforms these structures into PowerLoom code. The ApplicationInterface defines in addition the already mentioned KRSystemBackend which abstracts from the specific backend implementation. It is depicted in Fig. 8.8 to illustrate that a direct cyclic dependency between the subsystem and the PowerLoomBackend subsystem is avoided this way.

The following subsections will describe the single subsystems in detail.

8.3.1. Framework Core

Task of the Core subsystem is the implementation of finite structures and logical rules. The classes and interfaces of the implementation are accessible to all other subsystems of the ACC framework.

Finite Structures

Finite structures and basic operations on them are realized in the package finstr (*Finite Strutures*) of the framework core. The classes implemented in this package are depicted in the upper part of Fig. 8.9. For this diagram, as well as for all following diagrams describing the design of the framework, holds that it depicts only a partial view on the framework implementation focusing on the most relevant design aspects and omitting technical details.

A single finite structure as used in the framework is an instance of the class FiniteStructure. As in the formal definition (see Sec. 5.2), a finite structure consists of a universe and a set of relations. The universe of a finite structure is a set of entities, modelled as association between Universe and Entity. A Relation has an arity and refers to a corresponding sequence of types which serve as domains for the relation — e.g., a Relation instance for a binary relation must be connected by exactly two links to instances of Type. A relation consists of tuples, modelled by the association connecting Relation and Tuple. A tuple is a sequence of entities whereas the size of this sequence must be equal to the arity of the containing relation, and the sequence the entities' type must conform to the sequence of the domains.

Types are understood as unary relations, i.e. the value of arity attribute for a instance of Type is always one. The set of tuples of a type equals furthermore the set of entities being instances of this type; moreover, entities are always typed. Furthermore, subtyping between types is allowed. The boolean attribute isPrimitive indicates whether a type is primitive — entities being instances of primitive types do not have a unique identifier and are compared by the value they represent. Such entities are objects of the class PrimitiveValue.

The classes of this data model also define operations for the construction of finite structures. FiniteStructure, for example, has methods to add types and relations; Relation provides operation to add tuples to relations checking the conformance to the arity of the relation; Universe owns operations to create new entities as instances of existing types.

The util package of the Core subsystem contains, among other elements, two interfaces for traversing finite structures following the visitor pattern [GHJV95]. This pattern separates functions from the class structure they operate on in order to keep the single classes cohesive. New functions can easily be added this way without modifying elements of the class structure.

Functions are realized as *Visitor classes* implementing a *Visitor* interface. This interface defines for each class of the class structure a visit method as depicted for FiniteStructureVisitor in Fig. 8.9; the implementation by a concrete visitor class describes the behaviour of the visitor if an instance of that class is traversed. Furthermore, each class of the data model contained in the package finstr implements a *Visitable* interface such as VisitableElement. It defines an *accept* method whose implementation performs a callback at the passed FiniteStructureVisitor

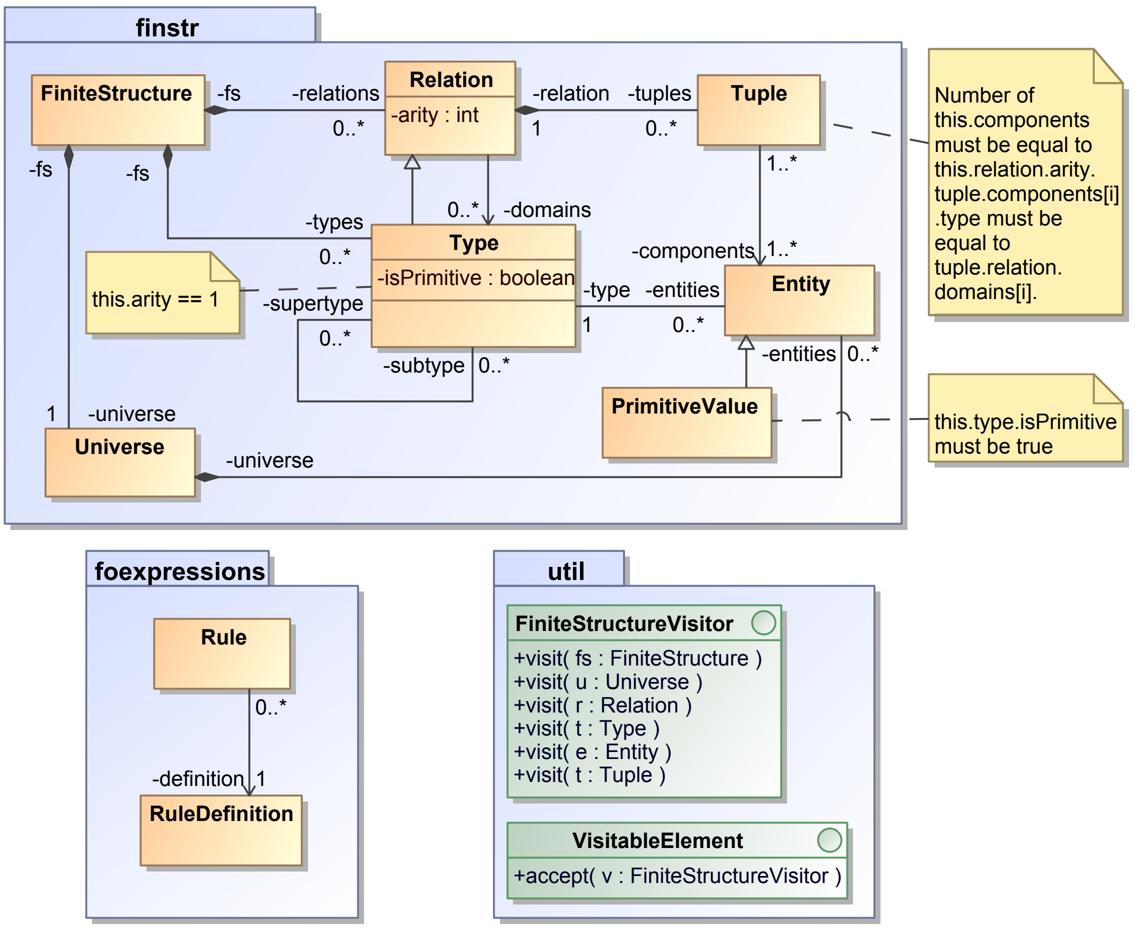


Figure 8.9: The Core subsystem of the ACC framework.

instance invoking its visit method. A visitor traversing an object structure calls the accept method of the object resulting in the callback of the visit method of the visitor; this dispatching ensures for polymorphic objects the call of the “correct” visit method without the need that the visitor knows the exact dynamic type of the object it traverses.

The usage of this pattern in the ACC framework is motivated by the separation of code generation functionality transforming finite structures into code of some knowledge representation systems from the finite structures themselves. This way, new code generators can easily be added without the need to modify the classes representing the elements of finite structures.

Rules

Intensional statements of models are represented by the classes in the package foexpression (see Fig. 8.9). The class RuleDefinition represents statements as used in the transformation definition for meta models containing free variables. The class Rule adds the possibility to add variable bindings; it combines the definition of a rule with a potentially partial binding of variables to specific entities of a finite structure.

The overall package is a placeholder in the prototypical implementation yet. Up till now,

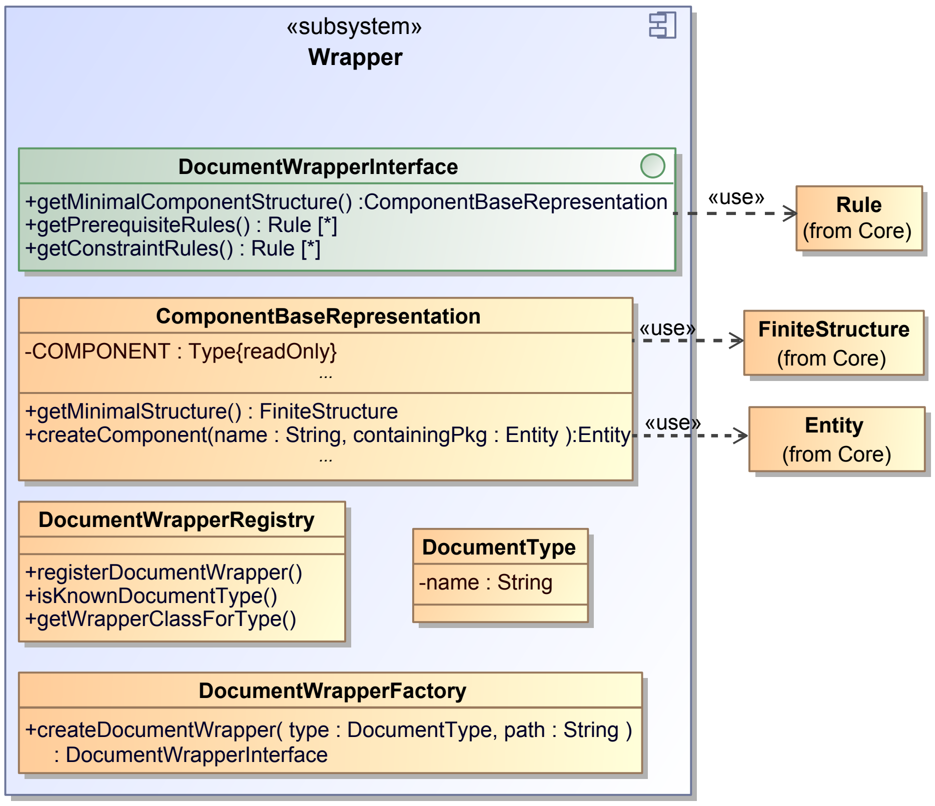

«subsystem»
Wrapper
DocumentWrapperInterface
+getMinimalComponentStructure() :ComponentBaseRepresentation
+getPrerequisiteRules() : Rule [*]
+getConstraintRules() : Rule [*]
«use»
Rule
(from Core)
ComponentBaseRepresentation
-COMPONENT : Type{readOnly}
...
+getMinimalStructure() : FiniteStructure
+createComponent(name : String, containingPkg : Entity):Entity
...
«use»
FiniteStructure
(from Core)
«use»
Entity
(from Core)
DocumentWrapperRegistry
+registerDocumentWrapper()
+isKnownDocumentType()
+getWrapperClassForType()
DocumentType
-name : String
DocumentWrapperFactory
+createDocumentWrapper(type : DocumentType, path : String)
: DocumentWrapperInterface

Figure 8.10: The Wrapper subsystem of the ACC framework.

rules are represented by strings only with special markers indicating variables. This means that the strings have to conform to the language of the underlying knowledge representation system. Instead, it would be more flexible to have an independent model of first-order logic expressions which means that the syntax of first-order expression had to be reflected by a data model, which will be integrated in the future.

8.3.2. Integrating Document Wrappers

The subsystem Wrapper is responsible for the integration of different meta models for the purpose of checking instances of them for compliance. The details of the subsystem are depicted in Fig. 8.10.

To be able to check a certain model for compliance, a wrapper for the model's meta model must exist. Such wrappers are implementations of the DocumentWrapper interface. The three most important methods that have to be implemented are:

- `getMinimalComponentStructure()`: Returns an object that represents the minimal τ_{CBSD}-structure for the extensional part of the wrapped model. It is assumed that only models being unambiguously factual (see Sec. 5.5) are wrapped, such that the minimal structure is unique and retrieved by the method.

- `getPrerequisiteRules()/getConstraintRules()`: Return the intensional rules that a model defines with the separation of prerequisite rules and constraint rules (see Sec. 8.1).

The returned minimal structure is an instance of ComponentBaseRepresentation which represents τ_{CBSD}-structure basing on FiniteStructure. It encapsulates a finite structure which can be retrieved by `getMinimalStructure()`. ComponentBaseRepresentation has a static initializer which reads a declarative description of τ_{CBSD} and generates a finite structure with empty universe and all relation (symbols) defined by τ_{CBSD}. When an instance of Component-BaseRepresentation is created, the static empty finite structure is copied to an instance field. Wrapper implementations can then use the created object to populate the universe according to the transformation definitions and the content of the model.

For this purpose, ComponentBaseRepresentation allows the developer of a wrapper implementation to access type and relation constants that reflect the relations according to τ_{CBSD}, such as COMPONENT as depicted in Fig. 8.10. Moreover, the class provides convenience operations to simplify the creation of the minimal system. Instead of specifying via the encapsulated FiniteStructure object that an entity of type COMPONENT is created and assigned to an entity of type PACKAGE by a tuple of the containsComponent relation, the developer can use methods like `createComponent(String compName, Entity containingPkg)`; these methods aggregate and encapsulate the single creation statements.

The rules returned by the methods defined in DocumentWrapperInterface are instances of the class Rule; each of such instances is assigned to a RuleDefinition instance reflecting the rule specification without any variable binding as described before. The source for rule definitions can be any source for strings, in the case of the wrapper for layered architectures, a simple text file is attached to wrapper instances during their initialization. It contains a comma-separated list of entries of the form

```
<meta model element>,<rule-def>,<rule-desc>
```

`<meta model element>` refers to the element of the meta model for which the rule is generated, `<rule-def>` contains the rule definition, and `<rule-desc>` is an informal description of the rule definition. The actual rule definition is PowerLoom query code which will be described partially in Sec. 8.3.4. For example, the entry

```
"Layer",
"(illegalLayerDependenciesFromGroup
     ?this ?toLayer ?srcElem ?trgElem)",
"Actual layer dependencies are not compliant
     with intended ones!"
```

defines the architectural rules for layers. From this file, the wrapper for layer models instantiates static RuleDefinition attributes for each entry. When analysing the wrapped model, the wrapper retrieves every Layer instance from the model and creates a Rule instance whose rule definition is the corresponding static rule definition and binds the variable `?this` to the entity representing the current layer. The generated rules are returned by the corresponding methods defined in DocumentWrapperInterface.

At the moment, there are two implementations of wrappers developed for the ArCh prototype. Both wrappers, for UML and for a specific meta model for layered architectures, are implemented based upon an EMF-generated API providing access to instances of the particular meta model. A model is programmatically traversed; the transformation definitions given in Sec. 6.2 and Sec. 6.3, respectively, are manually translated into code. Entities and tuples corresponding to the single extensional statements are created within the finite structure. Intensional statements are generated according to the rule definitions.

So far, the developer of a wrapper knows which interfaces to use and to implement. To integrate a wrapper into the ACC framework and to use it, the wrapper must be registered at the framework. This is done by using the DocumentWrapperRegistry whose most important methods are[2] (see also Fig. 8.10):

- `registerDocumentWrapper(DocumentType, Class<? extends DocumentWrapperInterface>)`: Registers the *class* of a document wrapper and assigns it to the document type given by the first parameter. DocumentType encapsulates simply a string naming the meta model, for example "Layer Architecture" in Fig. 8.2. The framework saves this way which class of wrappers is responsible for which kind of document.

- `boolean isKnownDocumentType(DocumentType)`: Returns whether the passed document type is already registered.

- `Class<? extends DocumentWrapperInterface> getWrapperClassForType(DocumentType docType)`: Returns the wrapper class responsible for the given document type.

All methods of DocumentWrapperRegistry are static and are usually called during the class initialization of wrapper implementations.

This piece of information held by the DocumentWrapperRegistry is used by the DocumentWrapperFactory. This class implements a factory [GHJV95] in a generic way: using the mapping between document types and wrapper classes persisted in the registry, its only method creates an instance of the wrapper class assigned to the given document type using reflection. The new wrapper instance is initialized with the path to the wrapped document/model.

8.3.3. The Application Interface

The application interface contains functions and data structures to access the framework from arbitrary tool frontends. Its functions comprise the configuration and management of compliance checks as well as the actual implementation of the checking functionality. The basic behaviour is simple: it retrieves the minimal structures as instances of the data model defined in the framework core from the participating models and their wrappers, and forwards them to the knowledge representation system in the background which transforms them into code for PowerLoom. Rule checks are treated the same way: rules are retrieved from the wrappers and forwarded for execution to the knowledge representation system. In the following, more details

[2]In Fig. 8.10, the signatures of the methods are omitted for layout reasons.

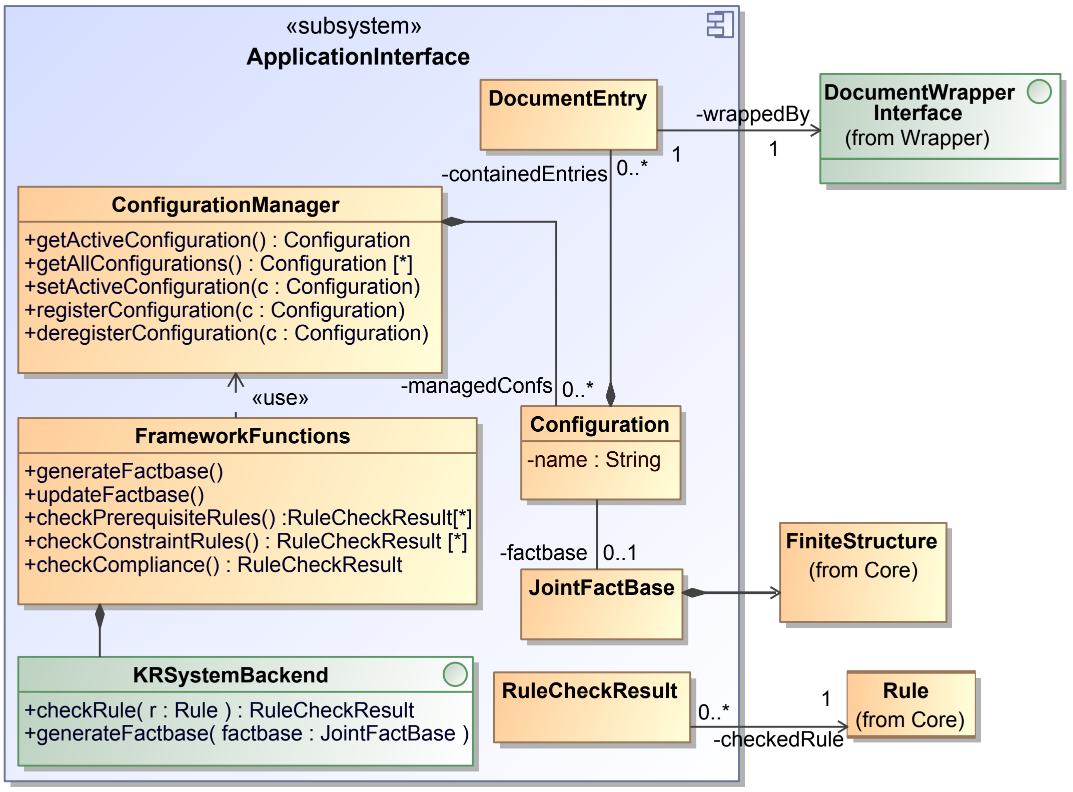

«subsystem»
ApplicationInterface
DocumentEntry
-wrappedBy
1
1
DocumentWrapper Interface
(from Wrapper)
-containedEntries
0..*
ConfigurationManager
+getActiveConfiguration() : Configuration
+getAllConfigurations() : Configuration [*]
+setActiveConfiguration(c : Configuration)
+registerConfiguration(c : Configuration)
+deregisterConfiguration(c : Configuration)
«use»
-managedConfs
0..*
FrameworkFunctions
+generateFactbase()
+updateFactbase()
+checkPrerequisiteRules() :RuleCheckResult[*]
+checkConstraintRules() : RuleCheckResult [*]
+checkCompliance() : RuleCheckResult
Configuration
-name : String
-factbase
0..1
JointFactBase
FiniteStructure
(from Core)
KRSystemBackend
+checkRule(r : Rule) : RuleCheckResult
+generateFactbase(factbase : JointFactBase)
RuleCheckResult
0..*
-checkedRule
1
Rule
(from Core)

Figure 8.11: The ApplicationInterface subsystem of the framework.

of the ApplicationInterface will be presented. Figure 8.11 shows the most important elements of this subsystem.

Compliance Check Configurations

In Sec. 8.1 we have already seen, that compliance checking using ArCh organizes the set of artefacts to be checked into *configurations*. The GUI of ArCh visualizes very closely the logical concepts that are provided by the ApplicationInterface of the ACC framework. As Fig. 8.11 shows, the ConfigurationManager administrates the set of compliance check configurations, realized by the class Configuration which includes registering and deregistering of configurations, activating them, and retrieving all available or the active configuration. A configuration is a set of instances of DocumentEntry. Each instance of this class represents an artefact that is part of a configuration and subject to compliance checking. It is associated with an instance of DocumentWrapperInterface providing the wrapper for the document represented by the document entry.

Joint Factbases

Compliance checks are executed as queries on a logical factbase representing the set of documents/models of the active configuration. Factbases are implemented by the class Joint-

Factbase which is a container for the single finite structures accessible by the corresponding document wrappers. It merges the finite structures of the single wrappers and provides different options to "fold" entities, i.e. entities of the single finite structures are considered equal according certain properties like the fully qualified name in τ_{CBSD}-structure such that only one entity in the merged structure is created for entities in the single structures meaning the same thing. Instances of this class are completely independent of the knowledge representation system that is used in the background.

Furthermore, JointFactbase keeps tracing information between model element and the entities of the merged structure to be able to refer to model elements for the presentation of rule check results. The user of compliance checking tools, e.g. a software architect, works with the models but not with the formal representation, such that the representation in terms of model elements is much more intuitive and user-friendly.

The joint factbase for a configuration is not automatically generated when a configuration is defined, a circumstance modelled by the multiplicity 0..1 of the association between Configuration and JointFactbase. Instead, the factbase is only generated if actively requested by the frontend via the functions of the class FrameworkFunctions.

Access to the Framework Application Logic

The class FrameworkFunctions provides static operations implementing the application logic of the main framework functionality, i. e. checking of compliance according to logical rules. The first task is hence to create factbases. For this purpose there exist two operations:

- `generateFactbase()` creates a joint factbase for the configuration being active at the moment the operation is called. If there is already a joint factbase, it will be overwritten.

- `updateFactbase()` creates also a factbase but checks in case of an existing factbase whether the encapsulated documents have been modified since the last generation and whether a new creation is necessary.

Both methods forward the internally generated JointFactbase object to the knowledge representation system which generates the language-specific knowledge base.

The actual checking functionality can be invoked by different functions:

- `checkIntensionalCompleteness()` checks for each minimal structure for the wrapped documents whether it is intensionally complete.

- `checkComponentAxioms()` checks for each minimal structure for the wrapped documents whether it is a component-based system. The axioms are directly loaded into PowerLoom as predefined queries (see below).

- `checkPrerequisiteRules()` and

- `checkConstraintRules()` executes the intensional rules retrieved by the participating wrappers as queries in PowerLoom.

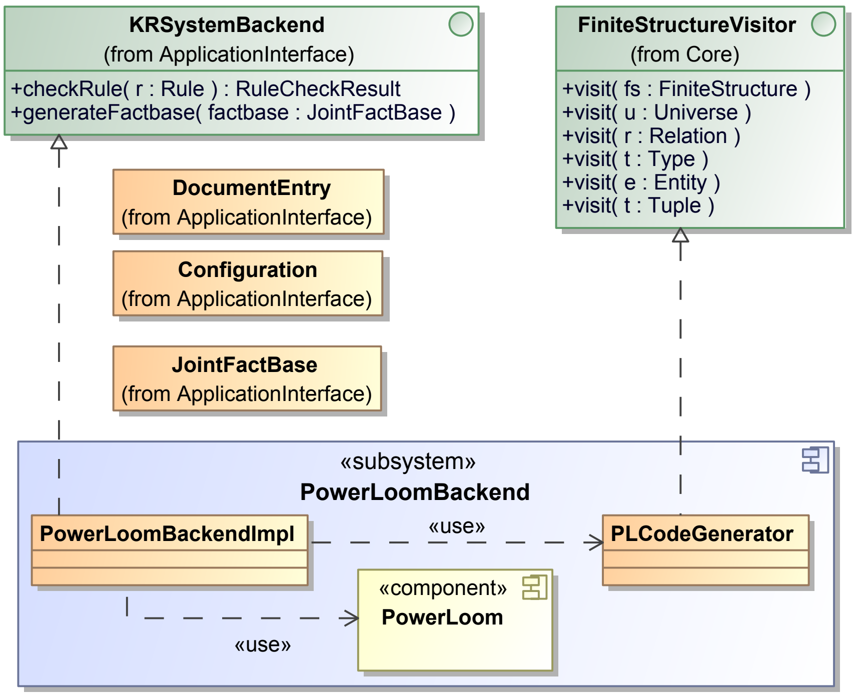


Figure 8.12: The PowerloomBackend subsystem of the ACC framework.

- `checkArchitecturalCompliance()` combines the previous operations to implement a generalized version of the compliance checking algorithm described in Listing 6.1. In contrast to the formal description, the implementation can deal with an arbitrary number of models instead of exactly two. Furthermore, it checks all rules but not only the architectural rules of the architectural model; if the precondition of intensional completeness holds, only non-local rules can become wrong in the joint factbase, hence a check of all rules is equivalent to the check of only architectural rules.

Each of these operations refers to the joint factbase of the active configuration. The class RuleCheckResult implements a logical representation of checking results; it refers to the rule that was executed as query by Powerloom and adds the bindings of free variables that lead to a violation of the rule. In contrast to Rule, which contains the bindings of variables that are defined before the query is executed, RuleCheckResult contains the bindings of the query result.

8.3.4. Integrating Knowledge Representation Systems

Knowledge representation systems are integrated into the framework by a system-specific implementation of the KRBackendInterface (see Fig. 8.11 and 8.12). This interface defines methods for the creation of knowledge bases and the checking of rules. The classes used as parameter and return types are defined in the ApplicationInterface subsystem of the ACC framework such that the operations defined there are independent of a specific knowledge representation system.

The implementation of the current version of the ACC framework is provided by the PowerLoomBackend subsystem as depicted in Fig. 8.12. Beside the implementation of the KRSystemBackend interface, the subsystem also implements a code generator for PowerLoom which is realized as visitor of finite structures according to the FiniteStructureVisitor interface defined in the subsystem Core (see Sec. 8.3.1). The generator is used by the PowerLoomBackendImpl class, e.g. to transform the fact bases given as instances of JointFactbase into PowerLoom knowledge bases. It generates concept and relation declarations for types and relations defined in the encapsulated finite structures, and transforms entities and tuples into facts denoted in the syntax of PowerLoom. More details of PowerLoom will be given in the following section.

8.4. PowerLoom

PowerLoom is a logic-based knowledge representation and reasoning system. It allows representing knowledge in a declarative way and provides diverse mechanisms to reason about knowledge and to make implicitly existing knowledge explicit. Knowledge is represented as entities and relations. A knowledge base can be queried interactively or via an Java API to retrieve data from knowledge bases that is either explicitly contained or can be derived by inference rules.

The PowerLoom system uses a full first-order logic language to express rules and queries with limited support for some higher-order constructs. The used syntax is the *Knowledge Interchange Format (KIF)* [GF92], which is basically a prefix notation of logical formulae. The principal undecidability of full first-order logics is addressed by limiting the reasoning mechanisms and making them adjustable.

The main reasoning mechanism is the calculus of natural deduction (see, e.g. [HR04]), which states how knowledge can be logically derived from existing explicit knowledge and rules in the knowledge base. One of the mentioned limitations in the implementation of PowerLoom is for example that the introduction of disjunctions as rule of the deduction calculus is not applied in the standard inference level of PowerLoom. However, the configuration can be customized and complemented by custom reasoners. Reasoning by cases is no realized at all — i.e., deducing a statement "C is true" from "C follows from A" and "C follows from B" and "A or B is true".

Both inference rules are more appropriate and required in application cases in which a proof must be searched in breadth, such as in theorem provers. In that application case, the goal is to infer a single proposition from an existing set of propositions, for example a set of axioms. In general, it is not decidable at any point of the proof whether a rule eliminating operators or introducing them is the best way; introduction rules add possible continuation points, while eliminating may get the proof closer to the final proposition. Hence, every kind of deduction is required.

The situation in checking architectural rules is different. On the one hand, there is a large set of propositions in form of facts of elementary terms. On the other hand, there are queries, representing architectural rules, which can be refined due to a relatively small set of implication rules, formally denoted by the definition of partial statements (e.g., *layerDepdency*, or *hasPublisherInterface*, see Sec. 6.2). These implications can be easily followed during "forward" deduction while the large fact base can be used to introduce more precise facts by introducing conjunctions or applying modus ponens. A search in breadth is not as important as in theorem provers, hence it seems that the missing deduction rules in PowerLoom will not affect the expressiveness of architectural rules too strongly.

Since inference in first-order logic languages has worst-case exponential complexity, PowerLoom has a number of built-in features addressing performance issues:

- Subsumption and a description classifier are used to efficiently compute the extension of relations [Mac94]. Originally developed in *description logics* [BCM+10], a description classifier computes subsumption relationships between descriptions of set of entities, i.e. intensional descriptions of types or relations. This way, the advantage of available type information can be applied for reasoning: a logical proposition does not need to be checked for all entities of the universe but only for those having the correct type.

- Integrated backward-chaining and forward-chaining inference are supported. Backward chaining starts with a logical expression as hypothesis, for which the reasoning systems looks for an inference rule with matching consequent. This is repeated with the antecedent instead of the hypothesis and repeated until the mechanism finds facts in the knowledge base that prove the correctness or the falsity of the hypothesis. PowerLoom can connect backward chaining and forward chaining and allows the declaration of rules as forward or backward only for efficiency reasons.

- Modules partition the overall knowledge base hierarchically and can keep reasoning focused. In PowerLoom, the overall knowledge base can be separated into different modules. Reasoning is in normal mode module-local, such that, if the knowledge base can be logically partitioned, reasoning can be reduced and efficiently executed on a limited subset of the overall knowledge base.

According to [Inf06], PowerLoom's reasoning is sound but not complete. This means that not every logical deduction theoretically possible can be executed by PowerLoom, but those it executes are correct. Furthermore, it always terminates due to a built-in recursion control.

Knowledge bases in PowerLoom are structured by *concepts* and *relations* whereas relations are understood in the mathematical sense and concepts compare to unary relations or types of entities. Before concepts and relations can actually be used in PowerLoom programs, they have to be declared. The declaration of concepts and relations can refer to supersetting relations, i.e. it can be specified that the set of instances of concepts or relations must be a subset of the instances of some other concepts or relations, respectively. The following code, for example, is generated by the PowerLoomBackend for the relations and types of the finite structure encapsulated in every instance of ComponentBaseRepresentation. They tell PowerLoom that there are the concepts of components and interfaces as well as provide and require relations between them.

```
(defconcept COMPONENT (CLASSIFIER))
(defconcept INTERFACE (OOCLASSIFIER))
(defrelation providesInterface ((?p1 COMPONENT) (?p2 INTERFACE)))
(defrelation requiresInterface ((?p1 COMPONENT) (?p2 INTERFACE)))
```

The PowerLoomBackend subsystem generates code for every relation symbol that is defined in τ_{CBSD} and additional symbols that are meta model-specific (e.g. for Layer). The defined concepts and relations can be used to assert *facts*; these represent explicit knowledge about the entities and their membership to concepts or relationships. In PowerLoom, facts are inserted into a knowledge base using the `assert` statement. The following code is generated for a finite structure modelling that the component Store is providing the interface StoreIf:

```
(assert (COMPONENT id_0))
(assert (INTERFACE id_1))
(assert (name id_0 "Store"))
(assert (name id_1 "StoreIf"))
(assert (providesInterface id_0 id_1))
```

Such code is generated for all of the entities and relation tuples encapsulated in a FiniteStructure instance when it is forwarded to the PowerLoomBackend for knowledge base generation.

As mentioned above, architectural compliance checking is realized as querying the logical knowledge base. The logical formulae defined in Sec. 6.2 are translated into PowerLoom code. Since the PowerLoom representation language supports all elements of first-order logic, the translation is straightforward. The left hand parts of the introduced definitions are declared as relations; the right hand part of the definitions are introduced as rules. Consider, for example, the following code implementing the definition of `illegalDependenciesFromGroup` given in (6.2):

```
(defrelation illegalDependenciesFromGroup
  ((?srcLayer GROUP)(?trgLayer GROUP)
   (?srcElem ELEMENT)(?trgElem ELEMENT)))
(defrule illegalLayerDependency_rhs
  (=> (and (GROUP ?l1)(GROUP ?l2)(ELEMENT ?e1)(ELEMENT ?e2)
           (not (= ?l1 ?l2))
           (not (isAllowedToUse ?l1 ?l2))
           (groupDependency ?l1 ?l2 ?e1 ?e2)
      )
     (illegalDependenciesFromGroup ?l1 ?l2 ?e1 ?e2)
  )
)
```

For user convenience, there are additional arguments compared to the formal definition in statement (6.2). This way, the inference algorithm of PowerLoom presents not only the layers causing a violation as answer of the query but also the components/interfaces[3]. The rule definition states that a illegal dependency between groups is implied (=>) if the groups are not equal, neither there exists a allowed usage relationships between them, and in addition there

[3]Nevertheless, a one-to-one translation of the formal definitions could also be done. PowerLoom is able to present proofs for query answers consisting of the deduction steps that have been made to answer the query; in this proof the variable bindings, and hence, which components or interfaces cause rule violations, would be visible.

	Test Series					
# Components	1	2	3	4	5	Average
10	0.13	0.16	0.17	0.16	0.16	0.156
20	0.25	0.31	0.22	0.28	0.27	0.266
50	0.53	0.52	0.50	0.52	0.55	0.524
100	1.08	1.05	1.06	1.05	1.00	1.048
500	18.20	18.58	18.17	18.03	17.61	18.118
1000	78.67	75.27	81.06	76.11	80.33	78.288
2500	514.66	503.19	503.55	502.94	510.09	506.886

Measured times in seconds

Table 8.1: Time consumption to check the architectural rules for layers in design models of different size measured in number of components.

is a dependency between the groups; `groupDependency` is defined according to statement (6.3) plus additional arguments for convenience reasons. Note, that for the correct usage of `not` the default open-world-semantics of PowerLoom is restricted; with open-world semantics, the non-provable fact is not considered false but to have an unknown truth value. In this, the fact that there is no isAllowedToUse link between two layers, does not mean that `(not (isAllowedToUse ?l1 ?l2))` is true, it must be explicitly asserted or implicitly inferred. The ACC framework initializes PowerLoom setting all relations and concepts to have closed-world semantics, which is ensured by asserting

```
(assert (closed rel))
(assert (closed c))
```

for all relations `rel` and concepts `c`.

The PowerLoom backend of the ACC framework is also initialized with the axioms of Φ_{CBSD} which are declared as rules (`defrule`). This way, knowledge bases can be checked whether they represent valid component-based systems.

Table 8.1 illustrates performance test series results for the PowerLoom implementation of the backend as used in ArCh. Tests included the implementation of the architectural rules for layer as described in Sec. 6.17. The checked models were UML design models and a layered architecture defining three layers with strict layering. The design models consisted of a defined number of components as depicted in the table. In addition to the components, the same number of interface were generated, as well as provide and require relations[4]; these connected components and interfaces randomly; moreover, components and interfaces were randomly assigned to packages, and hence indirectly to layers. Test were executed on a common desktop PC[5]. For comparison: a common installation of the Eclipse Ganymede edition comes with several hundred up to thousands of components whereas the system has about 18 million lines of code [Ske08].

[4]The number of relations were created according to the assumptions that each component is providing 1–2 interfaces and requiring 2 interfaces in average.

[5]Intel Core 2 CPU, 2.16GHz, 2GB RAM

The results show that the worst-case complexity of PowerLoom in querying does not affect checking the rule for layers. Time consumption exhibits a quadratic growth with the size of the design model. The absolute numbers, however, show that ArCh delivers checking results in a reasonable time at least for use cases in which checks are not permanently required. An application scenario for large systems could be to integrate architecture compliance checks into an automatic build process like nightly builds. As will be outlined in Sec. 9.2, tool support has to be technically optimized to deal with large systems like those represented by the larger models of the test series.

8.5. Conclusion and Summary

This chapter has presented the prototypical implementation of the concepts for architectural compliance checking. It has described the usage of ArCh, a simple architectural compliance checking prototype for Eclipse. Moreover, it has explained the overall architecture of ArCh and provided a detailed description of the reusable ACC framework as basis for general and more sophisticated tool support.

The ACC framework contains the main application logic of compliance checking and can be reused by the backends of arbitrary compliance checking tools. It realizes compliance checks as queries on a logical knowledge base implemented in PowerLoom, a powerful logical knowledge representation and reasoning system. The artefacts that need to be checked for compliance are integrated into the framework by wrappers implementing a wrapper interface. Wrappers provide the minimal τ_{CBSD}-structure of the model as well as the intensional rules that have to be checked for compliance between models. The framework retrieves those structures and rules from the wrappers and forwards them to the underlying PowerLoom system for the generation of knowledge bases and the execution of queries for compliance checking.

The architecture of the framework allows flexibility for tool support in the required dimensions. First of all, the underlying mechanism of a logical reasoning system provides high expressiveness regarding the formulation of architectural rules. It can be compared with query language-based approaches to compliance checking (see Sec. 3.2). The realization as framework to which meta model-specific wrappers can be connected allows us furthermore to easily add artefact types to compliance checking tools. Moreover, since architectural rules are declaratively added to the wrappers, single rules or the overall set of rules can easily be modified and adapted.

However, the effort of creating wrappers can still be reduced. First of all, instead of implementing the transformation definition for a given meta model to τ_{CBSD}-statements manually, model transformation techniques could be applied to specify the definition declaratively and execute them in a standardized way. Moreover, it should be investigated whether wrappers for meta models conforming to a common meta model like the MOF could be partially generated.

Chapter 9.

Conclusion

Contents

I am glad I did it, partly because it was well worth it, and chiefly because I shall never have to do it again.

Mark Twain

This chapter will discuss the results of this work and the proposed approach to architectural compliance checking. The contributions will be opposed to the limitations of the approach. Furthermore, an outlook to future research will be given. A summary will conclude the chapter and the overall thesis.

9.1. Discussion of Results

Chapter 2 concluded with the formulation of the research questions that motivated this thesis. The results of this thesis have to be measured by the answers it gives to that questions. We repeat the questions from Sec. 2.4 for the further discussion:

Research Question 1 (RQ 1): How can architectural compliance checking tool support be realized that is flexible with regard to

- the large number of different meta models that have to be considered in checking architectural compliance (RQ 1.1), and
- the variability and adaptability of different architectural rules that need to be checked (RQ 1.2)?

Research Question 2 (RQ 2): How can models of architectural design, detailed design and implementation be distinguished from each other to clearly define a term of architectural compliance, which, in order to provide checking tools, can be checked algorithmically?

The following subsections will discuss the contributions to these research questions and the limitations of the solution approach presented in this work.

9.1.1. Contributions

The delivered results of this thesis are the following:

- An formal framework has been developed, which
  - defines a formal representation of component-based systems and models of such systems,
  - defines the term of compliance of models/artefacts of the software development process based upon a formal classification of models/artefacts of architectural design, detailed design, and implementation, and
  - defines an algorithmic solution to checking compliance for a reasonable subset of models.
- Architectural rules have been specified for three "architectural" aspects: (a) layered systems as special case of specifying subsystems and relationships of allowed dependencies, (b) service-oriented interfaces, and (c) event-driven architectures. These aspects were separated according to the applied classification criterion into architectural and detailed design components.
- A transformation of standard UML into the formal representation has been defined for those parts of UML relevant for a case study, as well as for an architecture profile for UML and a design profile.
- Prototypical tool support has been implemented evaluating the developed concepts. This tool support consists on the one hand of an architectural compliance checking prototype (ArCh), which has been used to evaluate the developed architectural rules in parts. On the other hand, a reusable ACC framework has been developed, encapsulating the formal framework for easy reuse by more sophisticated compliance checking tools.

The following subsections will discuss the contributions of these results to answer the motivating research questions.

Contributions regarding Research Questions

The proposed approach addresses RQ 1.1 by the formal representation of models as first-order logic statements over τ_{CBSD}-structures. This formalism is used for the extensional content of models as well for the rules that might be contained restricting the further refinement of the described system by other artefacts/models of different types/meta models. The checking of rules is also applied to that logical representation of models, hence the definition of compliance and the proposed checking algorithm are independent of any specific meta model. To include instances of a certain meta model into the process of architectural compliance checking, a transformation definition must be specified describing how model elements are transformed into logical statements over τ_{CBSD}-structures. This has be done for standard UML, UML plus a profile for architecture description, and UML plus a profile for detailed design models.

This is a potentially powerful solution to the dimension of flexibility that is addressed by RQ 1.1. However, the provided flexibility must not only be judged by the possibility of integrating meta models into architectural compliance checking but also by the effort to do so. Considerations regarding the effort of integrating new meta models must be made at two distinct levels.

At the *conceptual level*, we can conclude that different meta models are supported by the approach as far as there can be given a meaningful transformation definition specifying how to transform an instance of the meta model into a set of corresponding τ_{CBSD}-statements. The effort of defining such a transformation is low in cases in which the modelling language itself contains component-based concepts and the mapping onto the τ_{CBSD}-ontology is simple. It is more difficult if artefacts should be considered, like object-oriented programming languages, not supporting such concepts. Although defining the mapping/transformation is harder in these cases, one should be aware that for the application of such languages/artefact types in component-based software development there is often a common, informal understanding of how to represent a component; for example, guidelines or technologies exist describing how to represent a component in Java code (e.g., EJB), or in MDSD transformation rules that generate code for components. The same holds for other concepts from component-based software development. This knowledge can be used for the definition of the transformation into τ_{CBSD}-statements and is, if it has only existed implicitly or informally, made explicit and formal to a certain degree.

At the *implementation* level, the effort of implementing a document wrapper must be achieved, which adds to the effort of defining the conceptual transformation. The wrappers so far are implemented by traversing the model instance and transforming it into τ_{CBSD}-structure/-statements, respectively, whereas traversing the model is implemented manually and specifically for each meta model. Although the transformation is basically implemented manually and procedurally, and the effort could probably be lowered by using model transformation techniques (see Sec. 8.3.2), the efforts of wrapper development seem reasonable: the implemented UML wrapper takes about 800 lines of code[1].

A comparison with the state of the art regarding RQ 1.1 is difficult, as already mentioned in Chapt. 3, because the effort of adapting those approaches is especially difficult to judge in cases, in which the extensibility of commercial closed-source tools had to be evaluated for comparison. However, as a direct result, it can be stated that the proposed is not restricted to check programming languages as all related approaches do; a UML wrapper has been realized

[1]Since the implementation does not realize the full transformation definition given in Sec. 6.3, this is a rough estimation based upon the existing code size and the covered percentage of transformation definitions.

and integrated into a working prototype (see Sec. 8.1).

As another important contribution to RQ 1.1, it should be noted that the proposed approach does not only allow flexibility with regard to artefact types that need to be checked for compliance *with* architectural rules but also for artefact types that *define* architectural rules *for* architectural compliance. In contrast to existing approaches, especially reflexion modelling-based approaches (see Sec. 3.4), rules can be attached to any modelling language that is used to describe software architectures. There is no need to introduce another notation in addition to a used ADL for compliance checking purposes.

RQ 1.2 is also addressed by the formal framework. It denotes architectural rules to be intensional and non-local first-order logic statements over τ_{CBSD}-structures. The ontology defined by τ_{CBSD} and Φ_{CBSD} describes component-based systems in great details, such that architectural rules have great expressiveness. It enables software architectures to describe rules restricting type structures like inheritance, the inner structure of types like components, interfaces, and classes, the configuration of component-based systems, and the control flow graphs of methods as specifications of component behaviour. The framework, nevertheless, allows extending this ontology by relation symbols to introduce new, for example architectural concepts, like layers. Extensions to τ_{CBSD} are considered in the definition of compliance and can be introduced technically by wrappers. As application of the concepts and for evaluation purposes, three architectural aspects have been expressed as architectural rules. A prototype that is able to check architectural rules as defined above has been implemented applying the logical knowledge representation system PowerLoom.

The (architectural) rules a model defines are derived from more abstract rules that are defined on meta model level, e.g. the architectural rule for layers in general is refined for a specific layer in a concrete model by binding some of the free variables by a fixed scheme (see Sec. 6.2.1). This means that the definition of the rules can be attached to the meta model for which they are defined but can be stored separated from it in the realization. The rules are that way easily adaptable and interchangeable which fits the requirement for easy customization.

Compared to existing approaches, the proposed approach combines the advantages of query language-based approaches and reflexion modelling. While the first have great expressiveness, their integration into model-based approaches is not given; reflexion modelling provides a high-level model to the degree it is required to express dependencies but have hence limited expressiveness. The proposed approach allows software architects to add full first-order logic rules in a customizable way to arbitrary high-level models of software systems. From the perspective of ADL, there is a practical contribution of the proposed approach in that far that it allows to extend the extensional description that existing ADL provide by intensional constraints requiring checking across a set of refining artefacts of arbitrary types.

It can be concluded that the proposed framework is a powerful solution to RQ 1 providing the base for flexible tool support.

To address RQ 2, the abstraction classification criterion defined by [EHK06] has been successfully refined and applied in the context of component-based systems. Architectural concepts, for example event-based architectures, are concepts defining intensional, non-local statements over τ_{CBSD}-structures. A model describing the application of an architectural concept in a system consists of such non-local statements but also has “its extension”, a set of extensional statement describing facts about systems conforming to the model.

The notion of compliance as stated in Definition 18 as part of the formal framework applies this separation and formally states that the extension of a model developed in the design process must adhere to the rules that models defined at the more abstract design levels. This is especially true for models of the detailed design and implementation that have to conform to architectural rules. The contribution of the formal framework in this respect is a formal definition of architectural rules as the "principles guiding the design and implementation of a system", based on the classification of [EHK06], and a formal notion of how it can be checked that these "guiding principles" [IEE00] are followed.

Moreover, the formal framework and its implementation as prototypical tool realize an algorithm for a practical relevant subset of models for which the checking can be done algorithmically. All of these contributions are evaluated by defining exemplary architectural rules and applying checks in a case study.

Additional Contributions

Another contribution of this thesis is a sharpened understanding of the role of software architecture as specification for design and implementation.

This contribution affects the way how architectures are described and understood. Existing approaches to architecture description, especially ADL, focus on extensional descriptions — i.e., the specification of components and connectors. Intensional statements are only used in approaches in which constraints for interaction protocols are specified whereas constraints restrict runtime structures but not static structures of refining models. All these issues lead to approaches that, due to the inability to specify constraints for the implementation level, provide extensional description techniques close to the implementation level themselves.

The proposed approach shows that the creation of a software architecture and its description can improve the development of software more efficiently if the software architecture is seen in context of the overall development process and its task to define the rules for the refinement. This especially includes that architectural rules are considered first-class elements of architectural descriptions which is not supported by any approach to software architecture description so far. Moreover, there is no integrated approach so far allowing the description of general architectural concepts, their application in the architecture of a specific system, together with the description of the architectural rules they define for the specific design and implementation.

The formal framework proposed in this thesis is one step to pave the road for architecture description techniques which are more focused on integrating "intensional aspects" into architecture modelling.

9.1.2. Limitations

As already mentioned in the previous section, the transformation of models into statements over τ_{CBSD}-structures becomes the harder the more the concepts of the modelling language differ from the concepts in τ_{CBSD}. This does not only explicitly refer to languages that are not component-based but must be mapped to component-oriented concepts; component models that do not assume axioms as strict as τ_{CBSD}/Φ_{CBSD} must be mapped with care, too; for example, UML allows ports that define provided and required interfaces at the same time, which is not allowed in τ_{CBSD}. It might be possible that the proposed ontology for component-based systems does not reflect single components models adequately enough. However, it must be stated, that the mechanism of compliance checking does not depend on the specific signature or the axiomatic system, and that both could be customized. Existing architectural rules, however, had to be adapted in certain cases.

The representation of behaviour as control flow graphs and the transformation of behavioural models into such structures might also limit the field of application of the proposed approach. The representation is strongly influenced from object-oriented systems in which behaviour is specified by implementing methods. A mapping of other behaviour specification techniques, for instance contract specifications, might be difficult to realize.

Limitations of the ontology that could be resolved by simple extensions are the consideration of concepts like multiplicities or generics which are ignored, for example, in the UML wrapper so far.

The formulation of logical rules and their expressiveness is always limited by the applied logic and the set of available predicates, i.e. the signature. First-order logics have proved expressive enough for the analysed architectural rules; moreover, PowerLoom provides good query performance for this formalism as discussed in Sec. 8.4. Nevertheless, the developed ontology τ_{CBSD}/Φ_{CBSD} lacks of certain expressiveness especially for rules/statements referring to the behavioural aspects.

In fact, concepts like program traces, call sequences, and other runtime constructs are missing. Consider, for example, the rule stating that transfer objects returned by services must be newly created objects (see Sec. 6.2.2). There are many possibilities for valid implementations of this: the service itself creates an object, a directly called method creates it, an indirect invoked method creates it, and so on. It is difficult to express these possibilities as disjunction over the valid control flow graphs that reflect these options. As already outlined in Sec. 6.2.1, it will be necessary to integrate techniques of static program analysis to add such information to the formal representation of component-based systems. So far, the expressiveness of the formalism is limited with regard to dynamic aspects of component-based systems.

Additionally, the set of available relation symbols reflects sort of the abstract syntax of architectural rules with the effect that its usage is inconvenient in certain cases. For example, to specify a method invocation in a rule that should have a certain parameter binding, the call itself, the binding of each single parameter, and the mapped formal and actual parameters must be referenced by single relations tuples of arity two (see, e.g. Sec. 6.2.2). Relation symbols of higher arity, for e.g. capturing a method invocation in a single tuple, should be defined for user convenience.

An issue that can be seen as disadvantage but also as possibility to customize the approach, is the definition of close relations (see Def. 16). The selection of these is in fact justified but open for interpretation. Different understandings might interpret additions to τ_{CBSD}-structures as modifications of existing elements that are not reflected in the set as defined. Hence, the actually selected relations are a point of discussion but do not affect the concepts of the overall approach.

However, it shows that there might be more reasonable solutions to define substructures in finite structures that are considered as "a whole" with regard to the definition of locality and intensionality.

The definition of compliance, which basically tests whether the extension of one model satisfies the constraints of another, is not applicable to an interesting task of compliance checking; in principal the selected formalism would enable us to check whether a detailed design concept in general contradicts or conforms to an architectural concept. This would mean to test an intensional, local statement for compliance with a non-local statement. The definition of compliance, however, refers to the extension of a model.

9.2. Future Work

There are several points for improvement and different directions to extend the presented work that are worth considering in future work.

The current way to integrate new meta models into the framework for compliance checking can surely be improved as outlined in Sec. 9.1. Given a common meta meta model like MOF, it could be worth to implement meta model-independent wrappers that are initialized with the specific meta model and the model instance to be wrapped. Declarative transformation techniques could be applied to reduce the effort of programming wrappers; however, the effort of defining correct declarative transformation definitions must always be compared with the effort of implementing wrappers manually.

Although the formal representation of models has been created according to the common concepts of component-based software and although it has been applied to describe the CoCoME case study, the transformation of system descriptions of different component models would be useful to further develop the ontology τ_{CBSD}/Φ_{CBSD} and to improve it. Moreover, efforts in this direction to overcome the most severe limitations should be undertaken, i.e. to allow architectural rules that are more expressive with regard to behavioural aspects.

To further improve the practical relevance of the approach, architectural rules (and also design rules) should be defined and collected in a reusable catalogue of rules. This catalogue could be used by developers of document wrappers and would reduce the barrier to apply architectural compliance checking at all. Starting point could be to formalize the rules that patterns of popular pattern catalogues define, e.g. [BMRS96].

Moreover, the formalization of architectural rules of commonly used architectural aspects would evaluate even more deeply whether the formalization and the expressiveness of the proposed approach cover a practical relevant subset of possible architectural rules.

Checking architectural compliance and detecting violations of rules are only the basis for exhaustive compliance management tool support. Since software systems are large and inherently complex, resolving violations and re-establishing compliance are difficult tasks, too. Experiences from consistency management show that automatic resolution of inconsistencies is only possible to a certain degree, and must often be complemented by possibilities to manually influence inconsistency repairing [BHLW07]. Approaches must make a trade-off between the degree of automating inconsistency repairing and the degree of human interaction in order to provide enough tool support but leave enough freedom to the users. For example, if a tool for architectural compliance checking detects a component in a layer causing a wrong dependency, should it try to move the component in the “correct” layer? If it should make suggestions how to solve the problem, what are meaningful solutions and how can they be detected?

Another logical continuation of this approach is the execution of architectural compliance checks at runtime. Although the main purpose is to detect architectural mismatches as soon as possible in the life cycle of a system, checking compliance late at runtime could further ensure architectural properties for systems that cannot be completely checked at design-time. For example, in dynamic-adaptive systems [Nie10], the components constituting the overall system at runtime are not even known at design time. This also means that architectural properties can be violated at runtime and cannot be checked at design time, or only approximated.

Further investigations can be made regarding the logical formalism that should be applied to architectural compliance checking. So far, first-order logics are applied whose principal undecidability leads to some restriction in the applied knowledge representation and reasoning system PowerLoom (see Sec. 8.4). There is always a trade-off between the expressiveness of the language and the efficiency of checking logical statements. It could be investigated for example whether description logics [BCM+10] as subset of first-order logics is able to cover a subset of architectural rules large enough to be used in practice.

Furthermore, it should be investigated whether other logics, such as approaches to temporal logics [GFR00] could be applied to address behavioural aspects of architectural rules more efficiently. Consider, for example, again the rules regarding services and transfer objects. Providing an appropriate signature τ'_{CBSD} as improvement of τ_{CBSD}, statements like "After returning from the service, the returned object is never again used as return value" could be possibly formulated as temporal logic statement and evaluated. The complicated statement over the structure of a control flow graph, or over newly added runtime concepts, would be unnecessary.

As outlined in the previous section, the main field for future work based upon this thesis is the description of software architecture with focus on the intensional constraints that a software architecture defines. The approach so far is only a temporary solution to the problem that current description techniques focus on extensional statements about systems. It allows attaching formally denoted intensional statements to existing modelling languages.

However, the principle goal should be that the software architect can use adequate architecture description languages providing means to both relevant things: specifying components and connectors (dependencies) *and* describing the applied architecture concepts including the architectural rules they define. Furthermore, it must be possible that software architects can rely on a repository of such concepts and architectural rules to reuse them and to easily customize them.

One of the most important points in this context is a proper integration of architectural patterns and reference architectures into software architecture descriptions. As seen for layers, patterns describe "abstract" architectural rules that are refined in the context of the software architecture of a specific system. An integrated approach has to define how architectural rules can be systematically derived from patterns and reference architectures.

The integration of the approach into MDSD has been made in that sense that a formalism has been proposed how to represent models for the purpose of compliance checking. So far, it has not been applied to use persistent mapping information of applied model transformations for compliance checking issues. For example, the method of the presented UML profile to map layers onto packages of a design model is to define corresponding elements (the dependency mapsToPkg, see Sec. 6.2.1) for software architectures, such that the mapping is an explicit part of an architectural model. In a model-driven approach, it is also thinkable that the mapping is realized implicitly by a transformation between architectural and design models, such that, if the transformation approach persists tracing information, the mapping should be considered by checking the corresponding architectural rules.

For all further developments holds that extensions to the approach should be applied to check compliance in realistic environments, i. e. for systems with realistic size and complexity. Since the main goal is to provide tool support in the context of architectural compliance with great flexibility, tools support will be enhanced and improved.

The existing prototypes with the functionality as described in this thesis should be improved with regard to usability in order to be even better accepted in more evaluating case studies. This refers, for example, to a more user-friendly way to define and modify architectural rules, either in a more intuitive syntax than KIF, or with support like syntax checking editors, integrated into architectural checking tools. Moreover, usability could be improved by better presentations of checking results, for example by highlighting the sources of violations directly in the participating models, comparable to the presentation options of reflexion modelling approaches.

Additionally, the existing ArCh prototype must be prepared to be applicable to realistic large-scale systems. Although a beta-version of the tool has been applied to a medium-sized system with about 130,000 lines of code [DDHR09], its prototypical status will hardly be able to deal with larger systems, since the finite structures, representing the models of such systems, are kept in memory. The persistent storage into a database, for example, is possible but not yet realized.

9.3. Summary

This chapter has concluded the results of this thesis realizing an approach to flexible tool support for architectural compliance checking based upon a logic-based formal framework. The framework defines an ontology of component-based systems that is used to represent models as logical statements about systems. This way, a formal distinction between software architecture and the refining steps of software development has been defined, resulting in a formal and algorithmically checkable definition of compliance.

The proposed approach allows to realize architecture compliance checking tools that are flexible with regard to supported meta models and to variability of architectural rules. An implementation framework allows such tools to be easily adapted to new meta models, such that existing compliance checking functionality can be easily enhanced to new models without the need to modify existing architectural rules. Moreover, it is not only possible to easily integrate meta models whose instances need to be compliant *with* some architectural model; also meta models for architecture description *defining* rules can be easily integrated.

The solution enables the specification and checking of architectural rules with the expressiveness of full first-order logics. In combination with the easy integration of models into the process of checking, the resulting basis for tool support allows significantly more powerful support in the future compared to the state of the art. This is also a result of the applied viewpoint on software architecture which emphasizes to understand software architecture not only as the description of components and dependencies but as the source of fundamental guidelines for the further development of a software system.

Appendix A.

The Axiom System Φ_{CBSD}

This section contains the axioms of Φ_{CBSD} as far as omitted in Chapt. 5. The axioms describe domain restrictions and multiplicity constraints for the relations of τ_{CBSD}. They are ordered in alphabetically by the names of the relation symbols.

Domain restriction for *actualParameter*:

$$\mathbf{CBSD_{55}} := \forall x : \forall y : \mathit{actualParameter}(x,y) \rightarrow \mathit{ParameterBinding}(x) \land \mathit{TypedElement}(y) \tag{A.1}$$

Multiplicity of *actualParameter* on second component for fixed first component is exactly one:

$$\mathbf{CBSD_{56}} := \forall x : \mathit{ParameterBinding}(x) \rightarrow \exists! y : \mathit{actualParameter}(x,y) \tag{A.2}$$

Domain restriction for *assignedTo*:

$$\mathbf{CBSD_{57}} := \forall x : \forall y : \mathit{assignedTo}(x,y) \rightarrow \mathit{InstanceCreation}(x) \land \mathit{ReferenceVariable}(y) \tag{A.3}$$

Multiplicity of *assignedTo* on second component for fixed first component is at most one:

$$\begin{aligned}\mathbf{CBSD_{58}} := {} & \forall x : \mathit{InstanceCreation}(x) \rightarrow ((\forall y : \neg\, \mathit{assignedTo}(x,y)) \lor \\ & (\exists! y : \mathit{assignedTo}(x,y)))\end{aligned} \tag{A.4}$$

AsynchronousInvocation subsets *MethodInvocation*:

$$\mathbf{CBSD_{59}} := \forall x : \mathit{AsynchronousInvocation}(x) \rightarrow \mathit{MethodInvocation}(x) \tag{A.5}$$

Boolean subsets *PrimitiveType*:

$$\mathbf{CBSD_{60}} := \forall x : \mathit{Boolean}(x) \rightarrow \mathit{PrimitiveType}(x) \tag{A.6}$$

Domain restriction for *boundParameter*:

$$\begin{aligned}\mathbf{CBSD_{61}} := {} & \forall x : \forall y : \mathit{boundParameter}(x,y) \rightarrow \mathit{MethodInvocation}(x) \land \\ & \mathit{ParameterBinding}(y)\end{aligned} \tag{A.7}$$

Multiplicity of *boundParameter* on on first component for fixed second component is exactly one:

$$\mathbf{CBSD_{62}} := \forall y : \mathit{ParameterBinding}(y) \rightarrow \exists! x : \mathit{boundParameter}(x,y) \tag{A.8}$$

Class subsets *OOClassifier*:
$$\mathbf{CBSD_{63}} \coloneqq \forall x : \mathit{Class}(x) \to \mathit{OOClassifier}(x) \tag{A.9}$$

Classifier subsets *Type*:
$$\mathbf{CBSD_{64}} \coloneqq \forall x : \mathit{Classifier}(x) \to \mathit{Type}(x) \tag{A.10}$$

Component subsets *Classifier*:
$$\mathbf{CBSD_{65}} \coloneqq \forall x : \mathit{Component}(x) \to \mathit{Classifier}(x) \tag{A.11}$$

ComponentInstanceCreation subsets *InstanceCreation*:
$$\mathbf{CBSD_{66}} \coloneqq \forall x : \mathit{ComponentInstanceCreation}(x) \to \mathit{InstanceCreation}(x) \tag{A.12}$$

ComponentInstanceDestruction subsets *InstanceDestruction*:
$$\mathbf{CBSD_{67}} \coloneqq \forall x : \mathit{ComponentInstanceDestruction}(x) \to \mathit{InstanceDestruction}(x) \tag{A.13}$$

Domain restriction for *configurationPart*:
$$\mathbf{CBSD_{68}} \coloneqq \forall x : \forall y : \mathit{configurationPart}(x, y) \to \mathit{SystemConfiguration}(x) \land \mathit{Part}(y) \tag{A.14}$$

Multiplicity of *configurationPart* on second component for fixed first component is at least one:
$$\mathbf{CBSD_{69}} \coloneqq \forall x : \mathit{SystemConfiguration}(x) \to \exists y : \mathit{configurationPart}(x, y) \tag{A.15}$$

Multiplicity of *configurationPart* on first component for fixed second component is at most one:
$$\begin{aligned}\mathbf{CBSD_{70}} \coloneqq \forall y : \mathit{Part}(y) \to ((\forall x : \neg\, \mathit{configurationPart}(x, y))\lor \\ (\exists! x : \mathit{configurationPart}(x, y)))\end{aligned} \tag{A.16}$$

Connector subsets *Part*:
$$\mathbf{CBSD_{71}} \coloneqq \forall x : \mathit{Connector}(x) \to \mathit{Part}(x) \tag{A.17}$$

Domain restriction for *connectorSource*:
$$\mathbf{CBSD_{72}} \coloneqq \forall x : \forall y : \mathit{connectorSource}(x, y) \to \mathit{Connector}(x) \land \mathit{SinglePart}(y) \tag{A.18}$$

Multiplicity of *connectorSource* on second component for fixed first component is exactly one:

$$\mathbf{CBSD_{73}} \coloneqq \forall x : \mathit{Connector}(x) \to \exists! y : \mathit{connectorSource}(x, y) \tag{A.19}$$

Domain restriction for *connectorTarget*:
$$\mathbf{CBSD_{74}} \coloneqq \forall x : \forall y : \mathit{connectorTarget}(x, y) \to \mathit{Connector}(x) \land \mathit{SinglePart}(y) \tag{A.20}$$

Multiplicity of *connectorTarget* on second component for fixed first component is exactly one:

$$\mathbf{CBSD_{75}} \coloneqq \forall x : \mathit{Connector}(x) \to \exists! y : \mathit{connectorTarget}(x, y) \tag{A.21}$$

ConstantSymbol subsets *ReferenceVariable*:

$$\mathbf{CBSD_{76}} := \forall x : \mathit{ConstantSymbol}(x) \rightarrow \mathit{ReferenceVariable}(x) \tag{A.22}$$

Domain restriction for *containsClass*:

$$\mathbf{CBSD_{77}} := \forall x : \forall y : \mathit{containsClass}(x, y) \rightarrow \mathit{Component}(x) \land \mathit{Class}(y) \tag{A.23}$$

Multiplicity of *containsClass* on on first component for fixed second component is exactly one:

$$\mathbf{CBSD_{78}} := \forall y : \mathit{Class}(y) \rightarrow \exists! x : \mathit{containsClass}(x, y) \tag{A.24}$$

Domain restriction for *containsComponent*:

$$\mathbf{CBSD_{79}} := \forall x : \forall y : \mathit{containsComponent}(x, y) \rightarrow \mathit{Component}(x) \land \mathit{Package}(y) \tag{A.25}$$

Multiplicity of *containsComponent* on second component for fixed first component is exactly one:

$$\mathbf{CBSD_{80}} := \forall x : \mathit{Component}(x) \rightarrow \exists! y : \mathit{containsComponent}(x, y) \tag{A.26}$$

Domain restriction for *containsEdge*:

$$\mathbf{CBSD_{81}} := \forall x : \forall y : \mathit{containsEdge}(x, y) \rightarrow \mathit{MethodBody}(x) \land \mathit{ControlFlowEdge}(y) \tag{A.27}$$

Multiplicity of *containsEdge* on on first component for fixed second component is exactly one:

$$\mathbf{CBSD_{82}} := \forall y : \mathit{ControlFlowEdge}(y) \rightarrow \exists! x : \mathit{containsEdge}(x, y) \tag{A.28}$$

Domain restriction for *containsInterface*:

$$\mathbf{CBSD_{83}} := \forall x : \forall y : \mathit{containsInterface}(x, y) \rightarrow \mathit{Package}(x) \land \mathit{Interface}(y) \tag{A.29}$$

Multiplicity of *containsInterface* on on first component for fixed second component is exactly one:

$$\mathbf{CBSD_{84}} := \forall y : \mathit{Interface}(y) \rightarrow \exists! x : \mathit{containsInterface}(x, y) \tag{A.30}$$

Domain restriction for *containsPkg*:

$$\mathbf{CBSD_{85}} := \forall x : \forall y : \mathit{containsPkg}(x, y) \rightarrow \mathit{Package}(x) \land \mathit{Package}(y) \tag{A.31}$$

Multiplicity of *containsPkg* on first component for fixed second component is at most one:

$$\mathbf{CBSD_{86}} := \forall y : \mathit{Package}(y) \rightarrow ((\forall x : \neg\, \mathit{containsPkg}(x, y)) \lor (\exists! x : \mathit{containsPkg}(x, y))) \tag{A.32}$$

Domain restriction for *containsStatement*:

$$\mathbf{CBSD_{87}} := \forall x : \forall y : \mathit{containsStatement}(x, y) \rightarrow \mathit{MethodBody}(x) \land \mathit{StatementNode}(y) \tag{A.33}$$

Multiplicity of *containsStatement* on on first component for fixed second component is exactly one:

$$\mathbf{CBSD_{88}} := \forall y : \mathit{StatementNode}(y) \rightarrow \exists! x : \mathit{containsStatement}(x, y) \tag{A.34}$$

Domain restriction for *declaresVariable*:

$$\mathbf{CBSD_{89}} := \forall x : \forall y : \mathit{declaresVariable}(x, y) \rightarrow \mathit{MethodBody}(x) \wedge \mathit{LocalVariable}(y) \tag{A.35}$$

Multiplicity of *declaresVariable* on on first component for fixed second component is exactly one:

$$\mathbf{CBSD_{90}} := \forall y : \mathit{LocalVariable}(y) \rightarrow \exists! x : \mathit{declaresVariable}(x, y) \tag{A.36}$$

Domain restriction for *definesMember*:

$$\mathbf{CBSD_{91}} := \forall x : \forall y : \mathit{definesMember}(x, y) \rightarrow \mathit{OOClassifier}(x) \wedge \mathit{Member}(y) \tag{A.37}$$

Multiplicity of *definesMember* on on first component for fixed second component is exactly one:

$$\mathbf{CBSD_{92}} := \forall y : \mathit{Member}(y) \rightarrow \exists! x : \mathit{definesMember}(x, y) \tag{A.38}$$

Domain restriction for *definesMethod*:

$$\mathbf{CBSD_{93}} := \forall x : \forall y : \mathit{definesMethod}(x, y) \rightarrow \mathit{Class}(x) \wedge \mathit{MethodBody}(y) \tag{A.39}$$

Multiplicity of *definesMethod* on on first component for fixed second component is exactly one:

$$\mathbf{CBSD_{94}} := \forall y : \mathit{MethodBody}(y) \rightarrow \exists! x : \mathit{definesMethod}(x, y) \tag{A.40}$$

Domain restriction for *definesSignature*:

$$\mathbf{CBSD_{95}} := \forall x : \forall y : \mathit{definesSignature}(x, y) \rightarrow \mathit{OOClassifier}(x) \wedge \mathit{Signature}(y) \tag{A.41}$$

Multiplicity of *definesSignature* on on first component for fixed second component is exactly one:

$$\mathbf{CBSD_{96}} := \forall y : \mathit{Signature}(y) \rightarrow \exists! x : \mathit{definesSignature}(x, y) \tag{A.42}$$

Domain restriction for *destructionReference*:

$$\begin{aligned}\mathbf{CBSD_{97}} := \forall x : \forall y : \mathit{destructionReference}(x, y) \rightarrow \mathit{InstanceDestruction}(x) \wedge \\ \mathit{ReferenceVariable}(y)\end{aligned} \tag{A.43}$$

Multiplicity of *destructionReference* on second component for fixed first component is exactly one:

$$\mathbf{CBSD_{98}} := \forall x : \mathit{InstanceDestruction}(x) \rightarrow \exists! y : \mathit{destructionReference}(x, y) \tag{A.44}$$

Domain restriction for *extends*:

$$\mathbf{CBSD_{99}} := \forall x : \forall y : \mathit{extends}(x, y) \rightarrow \mathit{OOClassifier}(x) \wedge \mathit{OOClassifier}(y) \tag{A.45}$$

Float subsets *PrimitiveType*:

$$\mathbf{CBSD_{100}} := \forall x : \mathit{Float}(x) \rightarrow \mathit{PrimitiveType}(x) \tag{A.46}$$

Domain restriction for *formalParameter*:

$$\mathbf{CBSD_{101}} := \forall x : \forall y : \mathit{formalParameter}(x, y) \rightarrow \mathit{ParameterBinding}(x) \land \mathit{Parameter}(y) \tag{A.47}$$

Multiplicity of *formalParameter* on second component for fixed first component is exactly one:

$$\mathbf{CBSD_{102}} := \forall x : \mathit{ParameterBinding}(x) \rightarrow \exists! y : \mathit{formalParameter}(x, y) \tag{A.48}$$

Domain restriction for *hasMember*:

$$\mathbf{CBSD_{103}} := \forall x : \forall y : \mathit{hasMember}(x, y) \rightarrow \mathit{OOClassifier}(x) \land \mathit{Member}(y) \tag{A.49}$$

Multiplicity of *hasMember* on first component for fixed second component is at least one:

$$\mathbf{CBSD_{104}} := \forall y : \mathit{Member}(y) \rightarrow \exists x : \mathit{hasMember}(x, y) \tag{A.50}$$

Domain restriction for *hasMethod*:

$$\mathbf{CBSD_{105}} := \forall x : \forall y : \mathit{hasMethod}(x, y) \rightarrow \mathit{Class}(x) \land \mathit{MethodBody}(y) \tag{A.51}$$

Multiplicity of *hasMethod* on first component for fixed second component is at least one:

$$\mathbf{CBSD_{106}} := \forall y : \mathit{MethodBody}(y) \rightarrow \exists x : \mathit{hasMethod}(x, y) \tag{A.52}$$

Domain restriction for *hasParameter*:

$$\mathbf{CBSD_{107}} := \forall x : \forall y : \mathit{hasParameter}(x, y) \rightarrow \mathit{Signature}(x) \land \mathit{Parameter}(y) \tag{A.53}$$

Multiplicity of *hasParameter* on on first component for fixed second component is exactly one:

$$\mathbf{CBSD_{108}} := \forall y : \mathit{Parameter}(y) \rightarrow \exists! x : \mathit{hasParameter}(x, y) \tag{A.54}$$

Domain restriction for *hasSignature*:

$$\mathbf{CBSD_{109}} := \forall x : \forall y : \mathit{hasSignature}(x, y) \rightarrow \mathit{OOClassifier}(x) \land \mathit{Signature}(y) \tag{A.55}$$

Multiplicity of *hasSignature* on first component for fixed second component is at least one:

$$\mathbf{CBSD_{110}} := \forall y : \mathit{Signature}(y) \rightarrow \exists x : \mathit{hasSignature}(x, y) \tag{A.56}$$

Domain restriction for *hasType*:

$$\mathbf{CBSD_{111}} := \forall x : \forall y : \mathit{hasType}(x, y) \rightarrow \mathit{TypedElement}(x) \land \mathit{Type}(y) \tag{A.57}$$

Multiplicity of *hasType* on second component for fixed first component is at most one:

$$\mathbf{CBSD_{112}} := \forall x : \mathit{TypedElement}(x) \rightarrow ((\forall y : \neg\, \mathit{hasType}(x, y)) \lor (\exists! y : \mathit{hasType}(x, y))) \tag{A.58}$$

Multiplicity of *hasType* on on first component for fixed second component is exactly one:

$$\mathbf{CBSD_{113}} := \forall y : \mathit{Type}(y) \rightarrow \exists! x : \mathit{hasType}(x, y) \tag{A.59}$$

Domain restriction for *implements*:

$$\mathbf{CBSD_{114}} \coloneqq \forall x : \forall y : \mathit{implements}(x, y) \rightarrow \mathit{Class}(x) \land \mathit{Interface}(y) \tag{A.60}$$

Domain restriction for *implementsSignature*:

$$\mathbf{CBSD_{115}} \coloneqq \forall x : \forall y : \mathit{implementsSignature}(x, y) \rightarrow \mathit{MethodBody}(x) \land \mathit{Signature}(y) \tag{A.61}$$

Multiplicity of *implementsSignature* on second component for fixed first component is exactly one:

$$\mathbf{CBSD_{116}} \coloneqq \forall x : \mathit{MethodBody}(x) \rightarrow \exists! y : \mathit{implementsSignature}(x, y) \tag{A.62}$$

Domain restriction for *inheritsMember*:

$$\mathbf{CBSD_{117}} \coloneqq \forall x : \forall y : \mathit{inheritsMember}(x, y) \rightarrow \mathit{OOClassifier}(x) \land \mathit{Member}(y) \tag{A.63}$$

Domain restriction for *inheritsMethod*:

$$\mathbf{CBSD_{118}} \coloneqq \forall x : \forall y : \mathit{inheritsMethod}(x, y) \rightarrow \mathit{Class}(x) \land \mathit{MethodBody}(y) \tag{A.64}$$

Domain restriction for *inheritsSignature*:

$$\mathbf{CBSD_{119}} \coloneqq \forall x : \forall y : \mathit{inheritsSignature}(x, y) \rightarrow \mathit{OOClassifier}(x) \land \mathit{Signature}(y) \tag{A.65}$$

InnerPart subsets *SinglePart*:

$$\mathbf{CBSD_{120}} \coloneqq \forall x : \mathit{InnerPart}(x) \rightarrow \mathit{SinglePart}(x) \tag{A.66}$$

InstanceCreation subsets *StatementNode*:

$$\mathbf{CBSD_{121}} \coloneqq \forall x : \mathit{InstanceCreation}(x) \rightarrow \mathit{StatementNode}(x) \tag{A.67}$$

InstanceDestruction subsets *StatementNode*:

$$\mathbf{CBSD_{122}} \coloneqq \forall x : \mathit{InstanceDestruction}(x) \rightarrow \mathit{StatementNode}(x) \tag{A.68}$$

Domain restriction for *instantiatedClassifier*:

$$\mathbf{CBSD_{123}} \coloneqq \forall x : \forall y : \mathit{instantiatedClassifier}(x, y) \rightarrow \mathit{InstanceCreation}(x) \land \mathit{Classifier}(y) \tag{A.69}$$

Multiplicity of *instantiatedClassifier* on second component for fixed first component is exactly one:

$$\mathbf{CBSD_{124}} \coloneqq \forall x : \mathit{InstanceCreation}(x) \rightarrow \exists! y : \mathit{instantiatedClassifier}(x, y) \tag{A.70}$$

Integer subsets *PrimitiveType*:

$$\mathbf{CBSD_{125}} \coloneqq \forall x : \mathit{Integer}(x) \rightarrow \mathit{PrimitiveType}(x) \tag{A.71}$$

Domain restriction for *invokedAt*:

$$\mathbf{CBSD_{126}} \coloneqq \forall x : \forall y : \mathit{invokedAt}(x, y) \rightarrow \mathit{MethodInvocation}(x) \land \mathit{ReferenceVariable}(y) \tag{A.72}$$

Multiplicity of *invokedAt* on second component for fixed first component is exactly one:

$$\mathbf{CBSD_{127}} \coloneqq \forall x : \mathit{MethodInvocation}(x) \rightarrow \exists! y : \mathit{invokedAt}(x, y) \tag{A.73}$$

Domain restriction for *invokedSignature*:

$$\mathbf{CBSD_{128}} \coloneqq \forall x : \forall y : \mathit{invokedSignature}(x, y) \rightarrow \mathit{MethodInvocation}(x) \land \mathit{Signature}(y) \tag{A.74}$$

Multiplicity of *invokedSignature* on second component for fixed first component is exactly one:

$$\mathbf{CBSD_{129}} \coloneqq \forall x : \mathit{MethodInvocation}(x) \rightarrow \exists! y : \mathit{invokedSignature}(x, y) \tag{A.75}$$

LocalVariable subsets *ReferenceVariable*:

$$\mathbf{CBSD_{130}} \coloneqq \forall x : \mathit{LocalVariable}(x) \rightarrow \mathit{ReferenceVariable}(x) \tag{A.76}$$

Interface subsets *OOClassifier*:

$$\mathbf{CBSD_{131}} \coloneqq \forall x : \mathit{Interface}(x) \rightarrow \mathit{OOClassifier}(x) \tag{A.77}$$

Domain restriction for *isDelegator*:

$$\mathbf{CBSD_{132}} \coloneqq \forall x : \forall y : \mathit{isDelegator}(x, y) \rightarrow \mathit{Connector}(x) \land \mathit{Boolean}(y) \tag{A.78}$$

Multiplicity of *isDelegator* on second component for fixed first component is exactly one:

$$\mathbf{CBSD_{133}} \coloneqq \forall x : \mathit{Connector}(x) \rightarrow \exists! y : \mathit{isDelegator}(x, y) \tag{A.79}$$

Member subsets *ReferenceVariable*:

$$\mathbf{CBSD_{134}} \coloneqq \forall x : \mathit{Member}(x) \rightarrow \mathit{ReferenceVariable}(x) \tag{A.80}$$

MethodInvocation subsets *StatementNode*:

$$\mathbf{CBSD_{135}} \coloneqq \forall x : \mathit{MethodInvocation}(x) \rightarrow \mathit{StatementNode}(x) \tag{A.81}$$

Domain restriction for *name*:

$$\mathbf{CBSD_{136}} \coloneqq \forall x : \forall y : \mathit{name}(x, y) \rightarrow \mathit{NamedElement}(x) \land \mathit{String}(y) \tag{A.82}$$

Multiplicity of *name* on second component for fixed first component is exactly one:

$$\mathbf{CBSD_{137}} \coloneqq \forall x : \mathit{NamedElement}(x) \rightarrow \exists! y : \mathit{name}(x, y) \tag{A.83}$$

Domain restriction for *nextParameter*:

$$\mathbf{CBSD_{138}} \coloneqq \forall x : \forall y : \mathit{nextParameter}(x, y) \rightarrow \mathit{Parameter}(x) \land \mathit{Parameter}(y) \tag{A.84}$$

Multiplicity of *nextParameter* on second component for fixed first component is at most one:

$$\textbf{CBSD}_{\textbf{139}} := \forall x : Parameter(x) \rightarrow ((\forall y : \neg\, nextParameter(x,y)) \vee (\exists! y : nextParameter(x,y))) \tag{A.85}$$

Multiplicity of *nextParameter* on first component for fixed second component is at most one:

$$\textbf{CBSD}_{\textbf{140}} := \forall y : Parameter(y) \rightarrow ((\forall x : \neg\, nextParameter(x,y)) \vee (\exists! x : nextParameter(x,y))) \tag{A.86}$$

ObjectCreation subsets *InstanceCreation*:

$$\textbf{CBSD}_{\textbf{141}} := \forall x : ObjectCreation(x) \rightarrow InstanceCreation(x) \tag{A.87}$$

ObjectDestruction subsets *InstanceDestruction*:

$$\textbf{CBSD}_{\textbf{142}} := \forall x : ObjectDestruction(x) \rightarrow InstanceDestruction(x) \tag{A.88}$$

OOClassifier subsets *Classifier*:

$$\textbf{CBSD}_{\textbf{143}} := \forall x : OOClassifier(x) \rightarrow Classifier(x) \tag{A.89}$$

Package subsets *NamedElement*:

$$\textbf{CBSD}_{\textbf{144}} := \forall x : Package(x) \rightarrow NamedElement(x) \tag{A.90}$$

Parameter subsets *ReferenceVariable*:

$$\textbf{CBSD}_{\textbf{145}} := \forall x : Parameter(x) \rightarrow ReferenceVariable(x) \tag{A.91}$$

Part subsets *ReferenceVariable*:

$$\textbf{CBSD}_{\textbf{146}} := \forall x : Part(x) \rightarrow ReferenceVariable(x) \tag{A.92}$$

Domain restriction for *passesValueFrom*:

$$\textbf{CBSD}_{\textbf{147}} := \forall x : \forall y : passesValueFrom(x,y) \rightarrow ReferenceAssignment(x) \wedge ReferenceVariable(y) \tag{A.93}$$

Multiplicity of *passesValueFrom* on second component for fixed first component is exactly one:

$$\textbf{CBSD}_{\textbf{148}} := \forall x : ReferenceAssignment(x) \rightarrow \exists! y : passesValueFrom(x,y) \tag{A.94}$$

Domain restriction for *passesValueTo*:

$$\textbf{CBSD}_{\textbf{149}} := \forall x : \forall y : passesValueTo(x,y) \rightarrow ReferenceAssignment(x) \wedge ReferenceVariable(y) \tag{A.95}$$

Multiplicity of *passesValueTo* on second component for fixed first component is exactly one:

$$\textbf{CBSD}_{\textbf{150}} := \forall x : ReferenceAssignment(x) \rightarrow \exists! y : passesValueTo(x,y) \tag{A.96}$$

Port subsets *SinglePart*:
$$\mathbf{CBSD_{151}} \coloneqq \forall x : \mathit{Port}(x) \rightarrow \mathit{SinglePart}(x) \tag{A.97}$$

PrimitiveType subsets *Type*:
$$\mathbf{CBSD_{152}} \coloneqq \forall x : \mathit{PrimitiveType}(x) \rightarrow \mathit{Type}(x) \tag{A.98}$$

Domain restriction for *providedInterface*:
$$\mathbf{CBSD_{153}} \coloneqq \forall x : \forall y : \mathit{providedInterface}(x,y) \rightarrow \mathit{ProvidedPort}(x) \wedge \mathit{Interface}(y) \tag{A.99}$$

Multiplicity of *providedInterface* on second component for fixed first component is exactly one:
$$\mathbf{CBSD_{154}} \coloneqq \forall x : \mathit{ProvidedPort}(x) \rightarrow \exists! y : \mathit{providedInterface}(x,y) \tag{A.100}$$

ProvidedPort subsets *Port*:
$$\mathbf{CBSD_{155}} \coloneqq \forall x : \mathit{ProvidedPort}(x) \rightarrow \mathit{Port}(x) \tag{A.101}$$

Domain restriction for *providesInterface*:
$$\mathbf{CBSD_{156}} \coloneqq \forall x : \forall y : \mathit{providesInterface}(x,y) \rightarrow \mathit{Component}(x) \wedge \mathit{Interface}(y) \tag{A.102}$$

Multiplicity of *providesInterface* on second component for fixed first component is at least one:
$$\mathbf{CBSD_{157}} \coloneqq \forall x : \mathit{Component}(x) \rightarrow \exists y : \mathit{providesInterface}(x,y) \tag{A.103}$$

Domain restriction for *qualifiedName*:
$$\mathbf{CBSD_{158}} \coloneqq \forall x : \forall y : \mathit{qualifiedName}(x,y) \rightarrow \mathit{NamedElement}(x) \wedge \mathit{String}(y) \tag{A.104}$$

Multiplicity of *qualifiedName* on second component for fixed first component is exactly one:
$$\mathbf{CBSD_{159}} \coloneqq \forall x : \mathit{NamedElement}(x) \rightarrow \exists! y : \mathit{qualifiedName}(x,y) \tag{A.105}$$

ReferenceAssignment subsets *StatementNode*:
$$\mathbf{CBSD_{160}} \coloneqq \forall x : \mathit{ReferenceAssignment}(x) \rightarrow \mathit{StatementNode}(x) \tag{A.106}$$

ReferenceVariable subsets *TypedElement*:
$$\mathbf{CBSD_{161}} \coloneqq \forall x : \mathit{ReferenceVariable}(x) \rightarrow \mathit{TypedElement}(x) \tag{A.107}$$

Domain restriction for *requiredInterface*:
$$\mathbf{CBSD_{162}} \coloneqq \forall x : \forall y : \mathit{requiredInterface}(x,y) \rightarrow \mathit{RequiredPort}(x) \wedge \mathit{Interface}(y) \tag{A.108}$$

Multiplicity of *requiredInterface* on second component for fixed first component is exactly one:
$$\mathbf{CBSD_{163}} \coloneqq \forall x : \mathit{RequiredPort}(x) \rightarrow \exists! y : \mathit{requiredInterface}(x,y) \tag{A.109}$$

RequiredPort subsets *Port*:

$$\mathbf{CBSD_{164}} := \forall x : \mathit{RequiredPort}(x) \rightarrow \mathit{Port}(x) \tag{A.110}$$

Domain restriction for *requiresInterface*:

$$\mathbf{CBSD_{165}} := \forall x : \forall y : \mathit{requiresInterface}(x,y) \rightarrow \mathit{Component}(x) \land \mathit{Interface}(y) \tag{A.111}$$

Return subsets *StatementNode*:

$$\mathbf{CBSD_{166}} := \forall x : \mathit{Return}(x) \rightarrow \mathit{StatementNode}(x) \tag{A.112}$$

Domain restriction for *returnedRef*:

$$\begin{aligned}\mathbf{CBSD_{167}} := {} & \forall x : \forall y : \mathit{returnedRef}(x,y) \rightarrow \mathit{Return}(x)\land \\ & \mathit{ReferenceVariable}(y)\end{aligned} \tag{A.113}$$

Multiplicity of *returnedRef* on second component for fixed first component is at most one:

$$\begin{aligned}\mathbf{CBSD_{168}} := {} & \forall x : \mathit{Return}(x) \rightarrow ((\forall y : \neg\, \mathit{returnedRef}(x,y))\lor \\ & (\exists! y : \mathit{returnedRef}(x,y)))\end{aligned} \tag{A.114}$$

Domain restriction for *returnedTo*:

$$\begin{aligned}\mathbf{CBSD_{169}} := {} & \forall x : \forall y : \mathit{returnedTo}(x,y) \rightarrow \mathit{SynchronousInvocation}(x)\land \\ & \mathit{ReferenceVariable}(y)\end{aligned} \tag{A.115}$$

Multiplicity of *returnedTo* on second component for fixed first component is at most one:

$$\begin{aligned}\mathbf{CBSD_{170}} := {} & \forall x : \mathit{SynchronousInvocation}(x) \rightarrow ((\forall y : \neg\, \mathit{returnedTo}(x,y))\lor \\ & (\exists! y : \mathit{returnedTo}(x,y)))\end{aligned} \tag{A.116}$$

Domain restriction for *returnedValue*:

$$\mathbf{CBSD_{171}} := \forall x : \forall y : \mathit{returnedValue}(x,y) \rightarrow \mathit{Return}(x) \land \mathit{ReferenceVariable}(y) \tag{A.117}$$

Multiplicity of *returnedValue* on second component for fixed first component is at most one:

$$\begin{aligned}\mathbf{CBSD_{172}} := {} & \forall x : \mathit{Return}(x) \rightarrow ((\forall y : \neg\, \mathit{returnedValue}(x,y))\lor \\ & (\exists! y : \mathit{returnedValue}(x,y)))\end{aligned} \tag{A.118}$$

RootNode subsets *StatementNode*:

$$\mathbf{CBSD_{173}} := \forall x : \mathit{RootNode}(x) \rightarrow \mathit{StatementNode}(x) \tag{A.119}$$

Domain restriction for *setsMember*:

$$\mathbf{CBSD_{174}} := \forall x : \forall y : \mathit{setsMember}(x,y) \rightarrow \mathit{Connector}(x) \land \mathit{Member}(y) \tag{A.120}$$

Multiplicity of *setsMember* on second component for fixed first component is at most one:

$$\begin{aligned}\mathbf{CBSD_{175}} := {} & \forall x : \mathit{Connector}(x) \rightarrow ((\forall y : \neg\, \mathit{setsMember}(x,y))\lor \\ & (\exists! y : \mathit{setsMember}(x,y)))\end{aligned} \tag{A.121}$$

SinglePart subsets *Part*:

$$\mathbf{CBSD_{176}} := \forall x : \mathit{SinglePart}(x) \rightarrow \mathit{Part}(x) \tag{A.122}$$

Signature subsets *TypedElement*:

$$\mathbf{CBSD_{177}} := \forall x : \mathit{Signature}(x) \rightarrow \mathit{TypedElement}(x) \tag{A.123}$$

Domain restriction for *sourceContext*:

$$\mathbf{CBSD_{178}} := \forall x : \forall y : \mathit{sourceContext}(x, y) \rightarrow \mathit{Connector}(x) \wedge \mathit{InnerPart}(y) \tag{A.124}$$

Multiplicity of *sourceContext* on second component for fixed first component is at most one:

$$\begin{aligned}\mathbf{CBSD_{179}} := {} & \forall x : \mathit{Connector}(x) \rightarrow ((\forall y : \neg\, \mathit{sourceContext}(x, y)) \vee \\ & (\exists! y : \mathit{sourceContext}(x, y)))\end{aligned} \tag{A.125}$$

Domain restriction for *srcNode*:

$$\mathbf{CBSD_{180}} := \forall x : \forall y : \mathit{srcNode}(x, y) \rightarrow \mathit{ControlFlowEdge}(x) \wedge \mathit{StatementNode}(y) \tag{A.126}$$

Multiplicity of *srcNode* on second component for fixed first component is exactly one:

$$\mathbf{CBSD_{181}} := \forall x : \mathit{ControlFlowEdge}(x) \rightarrow \exists! y : \mathit{srcNode}(x, y) \tag{A.127}$$

String subsets *PrimitiveType*:

$$\mathbf{CBSD_{182}} := \forall x : \mathit{String}(x) \rightarrow \mathit{PrimitiveType}(x) \tag{A.128}$$

Domain restriction for *successorNode*:

$$\mathbf{CBSD_{183}} := \forall x : \forall y : \mathit{successorNode}(x, y) \rightarrow \mathit{StatementNode}(x) \wedge \mathit{StatementNode}(y) \tag{A.129}$$

SynchronousInvocation subsets *MethodInvocation*:

$$\mathbf{CBSD_{184}} := \forall x : \mathit{SynchronousInvocation}(x) \rightarrow \mathit{MethodInvocation}(x) \tag{A.130}$$

Domain restriction for *targetContext*:

$$\mathbf{CBSD_{185}} := \forall x : \forall y : \mathit{targetContext}(x, y) \rightarrow \mathit{Connector}(x) \wedge \mathit{InnerPart}(y) \tag{A.131}$$

Multiplicity of *targetContext* on second component for fixed first component is at most one:

$$\begin{aligned}\mathbf{CBSD_{186}} := {} & \forall x : \mathit{Connector}(x) \rightarrow ((\forall y : \neg\, \mathit{targetContext}(x, y)) \vee \\ & (\exists! y : \mathit{targetContext}(x, y)))\end{aligned} \tag{A.132}$$

Domain restriction for *trgNode*:

$$\mathbf{CBSD_{187}} := \forall x : \forall y : \mathit{trgNode}(x, y) \rightarrow \mathit{ControlFlowEdge}(x) \wedge \mathit{StatementNode}(y) \tag{A.133}$$

Multiplicity of *trgNode* on second component for fixed first component is exactly one:

$$\mathbf{CBSD_{188}} := \forall x : \mathit{ControlFlowEdge}(x) \rightarrow \exists! y : \mathit{trgNode}(x, y) \tag{A.134}$$

Type subsets *NamedElement*:

$$\mathbf{CBSD_{189}} := \forall x : \mathit{Type}(x) \rightarrow \mathit{NamedElement}(x) \tag{A.135}$$

TypedElement subsets *NamedElement*:

$$\mathbf{CBSD_{190}} \coloneqq \forall x : \mathit{TypedElement}(x) \rightarrow \mathit{NamedElement}(x) \tag{A.136}$$

Appendix B.

The Architecture and Design Profiles for UML

B.1. The Architecture Profile

The architecture profile defined in this section contains stereotypes to extend standard UML with concepts to by able to model architectural concepts applied in the case study of this work (see Sec. 4.1.2 and Sec. 4.2). Figure B.1 depicts the overall profile. This section describes the single elements of the profile in alphabetical order.

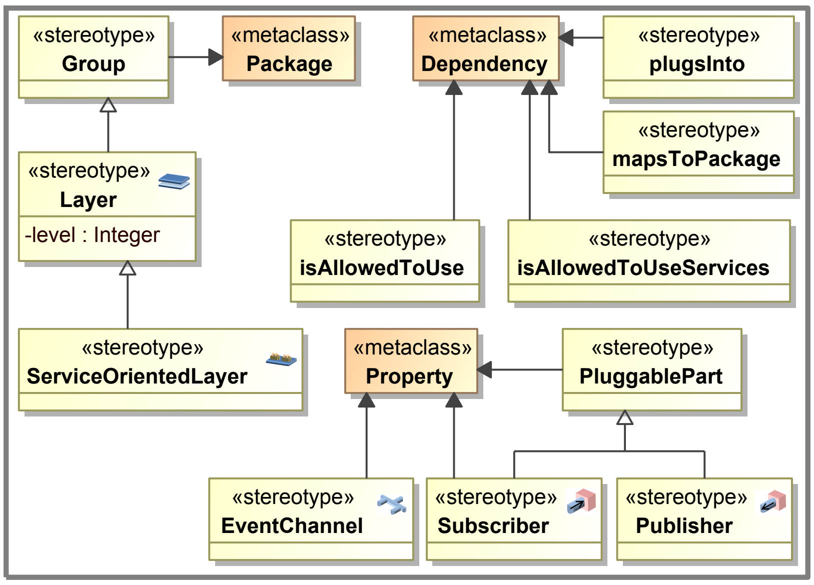


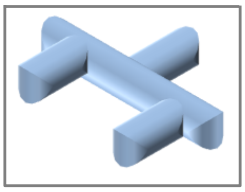
Figure B.1: The Architecture Profile.

B.1.1. EventChannel

Extended Meta Class

Property

Icon



Description

This stereotypes marks properties which represent an event channel in a system configuration or in a component.

Constraints and Transformation

- Instances of Property stereotyped with «EventChannel» must be part of a component which can optionally be stereotyped with «system ».

$$T^{UML,arch}(EventChannel) := \\ (\{EventChannelthis\}, \{hasECType(this), noBypassingOfChannel(this)\})$$

For refinement of architectural rules see (6-42) – (6.44).

B.1.2. Group

Extended Meta Class

Package

Icon

None.

Description

This stereotype marks “virtual” packages that logically group model elements. This means that the elements are not owned by such packages. Instead they are owned by other packages to which groups can be mapped. This way, it is possible to model groups or group hierarchies that cross-cut the primary package structure.

Constraints and Transformation

- The set of owned elements must be empty.

$$T^{UML,arch}(Group) := \{Group(this), \neg\, illegalDependenciesFromGroup(this)\}$$

For refinement of architectural rules see (6.1) – (6.13).

B.1.3. isAllowedToUse

Extended Meta Class

Dependency

Icon

None.

Description

This stereotypes expresses that is allowed to have dependencies between the group at the source of the relationship towards the target of the relationship. Dependencies between groups manifest as dependencies between packages to which the participating groups are mapped by «mapsToPkg».

Constraints and Transformation

- Source as well as target of the dependency must be stereotyped with «Group» or with a specialization of it.

$$T^{UML,arch}(isAllowedToUse) := (\{isAllowedToUse(this.source, this.target)\}, \emptyset)$$

B.1.4. isAllowedToUseServices

Extended Meta Class

Dependency, specializes «isAllowedToUse».

Icon

None.

Description

Marks dependencies towards service-oriented layers expressing that the group at the source of the dependency may only use the explicitly marked services of the layer.

Constraints and Transformation

- Source of dependency must be a package stereotyped with «Group» or with one of its specializations.
- Target of dependency must be a package stereotyped with «ServiceOrientedLayer».

$$T^{UML,arch}(isAllowedToUseServices) := \\ (\{isAllowedToUseServices(this.source, this.target)\}, \emptyset)$$

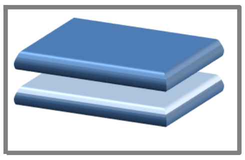
The transformation definition of «isAllowedToUse» also applies to instances stereotyped with «isAllowedToUseServices».

B.1.5. Layer

Extended Meta Class

Package, specialization of Group.

Icon



Description

Layers are a special form of groups. They can be used to structure the model content hierarchically. A package stereotyped with «Layer» defines a layer level (attribute level, which defines a hierarchical order between layers regarding «isAllowedToUse» relationships.

Constraints and Transformation

- For all outgoing «isAllowedToUse» dependencies of l has to hold: if the target m is stereotyped with «Layer», then the level of m must be lower than that of l.

$$T^{UML,arch}(Layer) := (\{Layer(this), level(this, this.level)\}, \emptyset)$$

The transformation of «Group» also applies to instances of «Layer».

B.1.6. mapsToPkg

Extended Meta Class

Dependency

Icon

None.

Description

This stereotype marks dependency between groups and “non-virtual” packages expressing to which packages a group, or one of its specializations, is mapped.

Constraints and Transformation

- Source of dependency must be stereotyped with «Group» or a specialization like Layer.
- Target of dependency may not be stereotyped with «Group» or a specialization like Layer.

$T^{UML,arch}(mapsToPkg) := (\{mapsToPkg(this.source, this.target)\}, \emptyset)$

B.1.7. PluggablePart

Extended Meta Class

Property

Icon

None.

Description

This stereotype is an abstract super class for instantiable stereotypes representing properties that can be plugged into event channels indicating that they are either able to receive or publish events (or both).

Constraints and Transformation

- Property must be part of a component, optionally stereotyped with «system».

$T^{UML,arch}(PluggablePart) := (\{PluggablePart(this)\}, \emptyset)$

B.1.8. plugsInto

Extended Meta Class

Dependency

Icon

None.

Description

Connects pluggable parts with an EventChannel instance modelling that a pluggable part is able to publish and receive events, depending on the instantiated specialization, via the connected channel.

Constraints and Transformation

- Source of dependency must be stereotyped with «PluggablePart» or one of its specializations.
- Target of dependency must be stereotyped with «EventChannel» or one of its specializations.

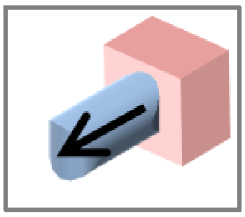
$T^{UML,arch}(plugsInto) := (\{plugsInto(this.source, this.target)\}, \emptyset)$

B.1.9. Publisher

Extended Meta Class

Property, specializes «PluggablePart».

Icon



Description

Marks properties that are publishers of events at one or more connected event channels.

Constraints and Transformation

$T^{UML,arch}(Publisher) :=(\{Publisher(this)\}, \{hasPublisherType(this)\})$

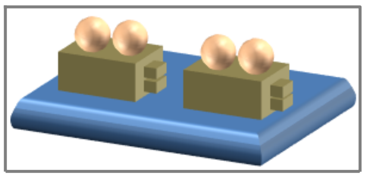
The transformation definition of «PluggablePart» also applies to instances stereotyped with «PluggablePart».
For refinement of architectural rules see (6.42) and (6.43)

B.1.10. ServiceOrientedLayer

Extended Meta Class

Package, specializes «Layer».

Icon



Description

This stereotype marks a special variant of layers providing services (see App. B.2) to clients of the layer.

Constraints and Transformation

$$T^{UML,arch}(ServiceOrientedLayer) := \\ (\{ServiceOrientedLayer(this)\}, \{onlyServicesExternallyUsed(this)\})$$

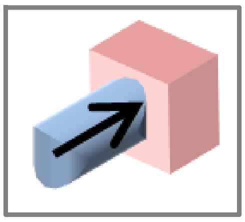
The transformation definitions of «Group» and «Layer» also apply to instances stereotyped with «Subscriber».
For refinement of architectural rules see (6.25) – (6.27).

B.1.11. Subscriber

Extended Meta Class

Property, specializes «PluggablePart».

Icon



Description

Marks properties that are subscribers to events at one or more connected event channels.

Constraints and Transformation

$$T^{UML,arch}(Subscriber) := (\{Subscriber(this)\}, \{hasSubscriberType(this)\})$$

The transformation definition of «PluggablePart» also applies to instances stereotyped with «Subscriber».
For refinement of architectural rules see (6.42) and (6.43).

B.2. The Design Profile

The design profile covers the parts of the considered architectural aspects that need to be expressed at the detailed design level. Fig. B.2 shows an overview of all available stereotypes defined by the profile. They will be described in detail in the following subsections.

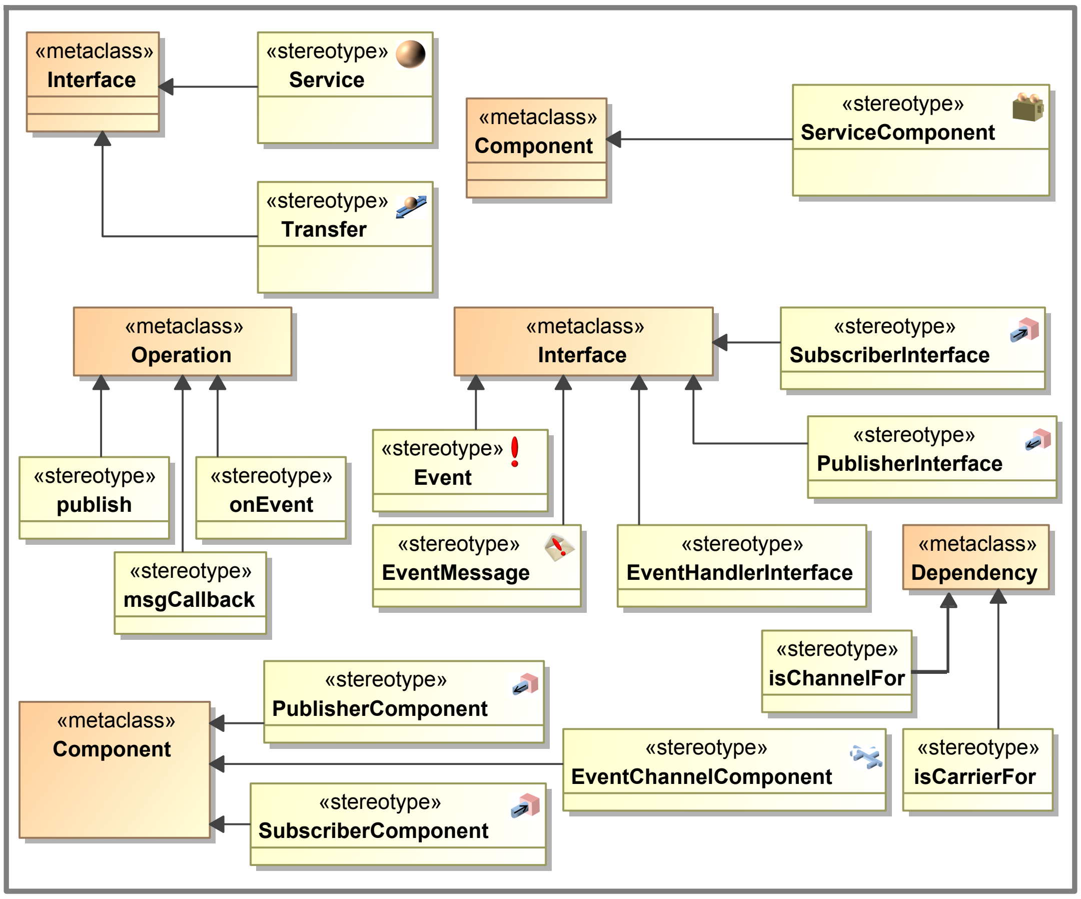

«metaclass» Interface
«stereotype» Service
«metaclass» Component
«stereotype» ServiceComponent
«stereotype» Transfer
«metaclass» Operation
«metaclass» Interface
«stereotype» SubscriberInterface
«stereotype» PublisherInterface
«stereotype» publish
«stereotype» onEvent
«stereotype» msgCallback
«stereotype» Event
«stereotype» EventMessage
«stereotype» EventHandlerInterface
«metaclass» Dependency
«stereotype» isChannelFor
«stereotype» PublisherComponent
«metaclass» Component
«stereotype» EventChannelComponent
«stereotype» isCarrierFor
«stereotype» SubscriberComponent

Figure B.2: The Design Profile.

B.2.1. Event

Extended Meta Class

Interface

Icon

Description

Marks interfaces that represent events.

Constraints and Transformation

$T^{UML,design}(Event) :=(\{Event(this)\}, \emptyset)$

B.2.2. EventChannelComponent

Extended Meta Class

Component

Icon

Description

This stereotype marks components that provide the functionality required to implement event channels. This means that (a) it provides a port at which event can be published, (b) it requires ports at which subscribers can be informed about events, and (c) it forwards events correctly between publishers and subscribers.

Constraints and Transformation

$$T^{UML,design}(EventChannelComponent) := \\ (\{EventChannelComponent(this)\}, \{isValidEventChannelComp(this)\})$$

For refinement of design rules see (6.36), (6.37), and (6.40).

B.2.3. EventHandlerInterface

Extended Meta Class

Interface

Icon

None.

Description

This stereotype marks interfaces responsible for processing events at subscribers.

Constraints and Transformation

$$\begin{aligned} T^{UML,design}&(EventHandlerInterface) := \\ &(\{EventHandlerInterface\}, \{isValidEventHandlerIf(this)\}) \end{aligned}$$

For refinement of design rules see (6.29), (6.34), and (6.35).

B.2.4. EventMessage

Extended Meta Class

Interface

Icon

Description

Marks interfaces that are able to transport events via an event channel — this means, that an interface marked as «EventMessage» is somehow able to save or forward objects whose type is implementing a «Event» interface.

Constraints and Transformation

$$T^{UML,design}(EventMessage) :=(\{EventMessage(this)\}, \{isEventContainer(this)\})$$

For refinement of design rules see (6.28).

B.2.5. isCarrierFor

Extended Meta Class

Dependency

Icon

None.

Description

Represent the information which events can be transported by an instance of «EventMessage». It connects this instances with those event that can be processed by the event message for the purpose of forwarding by an event channel. While interfaces stereotyped with «EventMessage» are mostly elements of a technical messaging framework, «Event» instances are mostly application-specific. «isCarrierFor» connects both levels.

Constraints and Transformation

- Source of dependency must be an interface stereotyped with «EventMessage».
- Target of dependency must be an interface stereotyped with «Event».

$T^{UML,design}(isCarrierFor) := (\{isCarrierFor(this.source, this.target)\}, \emptyset)$

B.2.6. isChannelFor

Extended Meta Class

Dependency

Icon

None.

Description

This stereotypes relates properties playing the role of event channels and the event that are actually forwarded by this specific event channel.

Constraints and Transformation

- Source of dependency must be property whose type is stereotyped with «EventChannelComponent».
- Target of dependency must stereotyped with «Event».

$T^{UML,design}(isChannelFor) := (\{isChannelFor(this.source, this.target)\}, \emptyset)$

B.2.7. msgCallback

Extended Meta Class

Operation

Icon

None.

Description

Marks methods of subscribers that serve as callback for the the event channel

Constraints and Transformation

$$T^{UML,design}(onMsgCallback) := \\ (\{onMsgCallback(this)\}, \{hasMsgCallbackSig(this)\})$$

For refinement of design rules see (6.31)

B.2.8. onEvent

Extended Meta Class

Operation

Icon

None.

Description

Marks operations that implement the reacting behaviour of event subscribers to a certain event.

Constraints and Transformation

$$T^{UML,design}(onEvent) := (\{onEvent(this)\}, \{hasOnEventSig(this)\})$$

For refinement of design rules see (6.35).

B.2.9. publish

Extended Meta Class

Operation

Icon

None.

Description

Marks operation that can be called to publish event. Mostly provided by event channels or other event distribution concepts.

Constraints and Transformation

$T^{UML,design}(publish) :=(\{publish(this\}, \{hasPublishSig(this)\})$

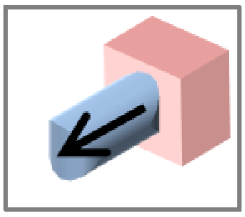
For refinement of design rules see (6.33).

B.2.10. PublisherComponent

Extended Meta Class

Component

Icon



Description

This stereotype marks components that are going to publish event at event channels. They require a publisher interface and are connected to an event channel at runtime providing such an interface.

Constraints and Transformation

$$\begin{aligned} T^{UML,design}&(PublisherComponent) := \\ &(\{PublisherComponent(this)\}, \{isValidPublisherComp(this)\}) \end{aligned} \tag{B.1}$$

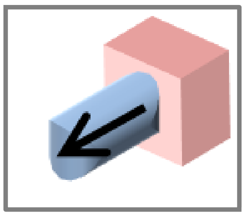
For refinement of design rules see (6.36) and (6.39).

B.2.11. PublisherInterface

Extended Meta Class

Interface

Icon



Description

Marks interfaces used for the purpose of publishing events. Such interface define at least one operation stereotyped with «publish».

Constraints and Transformation

$$T^{UML,design}(PublisherInterface) :=$$
$$(\{PublisherInterface(this)\}, \{isValidPublisherIf(this)\})$$

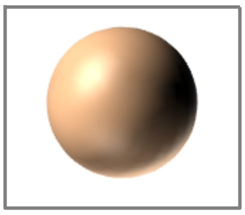
For refinement of design rules see (6.29), (6.32), and (6.33).

B.2.12. Service

Extended Meta Class

Interface

Icon



Description

This stereotype marks service interfaces, i.e. interfaces providing only operations with copy semantics (see Sec. 4.2.2 and 6.2.2.

Constraints and Transformation

$$T^{UML,design}(Service) := (\{Service(this)\}, \{hasServiceMethodsOnly(this)\})$$

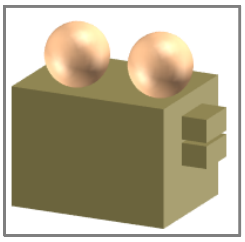
For refinement of design rules see (6.17) – (6-19).

B.2.13. ServiceComponent

Extended Meta Class

Component

Icon



Description

Marks components providing services, or more precisely, providing at least one interface stereotyped with «Service».

Constraints and Transformation

$$T^{UML,design}(ServiceComponent) := \\ (\{ServiceComponent(this)\}, \{providesService(this, this.provided)\})$$

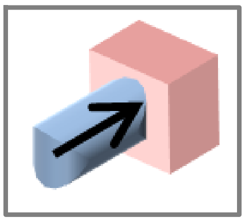
For refinement of design rules see (6.20) – (6.24).

B.2.14. SubscriberComponent

Extended Meta Class

Component

Icon



Description

Marks components realizing subscribers. Such component provide a subscriber interface and an event handler interface to react to events forwarded by event channels (see «EventHandler-Interface» and «SubscriberInterface»).

Constraints and Transformation

$$T^{UML,design}(SubscriberComponent) := \\ (\{SubscriberComponent\}, \{isValidSubscriberComp(this)\})$$

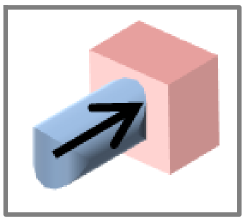
For refinement of design rules see (6.36) – (6.38).

B.2.15. SubscriberInterface

Extended Meta Class

Interface

Icon



Description

Marks interfaces providing subscriber functionality by providing a message call back. All subscribers need to implement an interface stereotyped with «SubscriberInterface».

Constraints and Transformation

$$T^{UML,design}(SubscriberInterface) := \\ (\{SubscriberInterface(this)\}, \{isValidSubscriberIf(this)\}) \qquad (B.2)$$

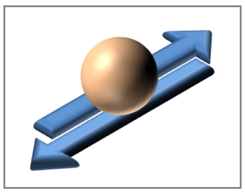
For refinement of design rules see (6.29) – (6.30).

B.2.16. Transfer

Extended Meta Class

Interface

Icon



Description

Marks interfaces for transfer objects. Transfer objects instantiate classes that implement an interface stereotyped with «Transfer».

Constraints and Transformation

$$T^{UML,design}(Transfer) := \quad (\{Transfer(this\}, \{onlyLegalMembers(this), onlyGettersAndSetters(this)\})$$

For refinement of design rules see (6.14) – (6.16).

Appendix C.

The Reference Design Model of CoCoME

The Figures C.1 and C.2 illustrate the grouping of components and interfaces in packages. Fig. C.1 focuses on the components and interfaces of the inventory subsystem. The components and interfaces of the cash desk subsystem are depicted in Fig. C.2
The system configuration of the CoCoME system is depicted in Fig. C.3 and Fig. C.4 whereas the first shows the parts constituting the inventory subsystem; the latter shows the cash desk system. The figures provide two partial views on the same single system configuration.
The Fig. C.5 - C.7 show the components of the Data subpackage in the inventory system. They provide the types for the parts that constitute the data layer of the system at runtime.
The Fig. C.8 - C.13 show the components of the Application subpackage in the inventory system. They provide the types for the parts that constitute the application layer of the system at runtime.
The Fig. C.14 - C.21 show the components of the cash desk subsystem of CoCoME.
Figures C.22 - C.24 show representative sequence diagrams.

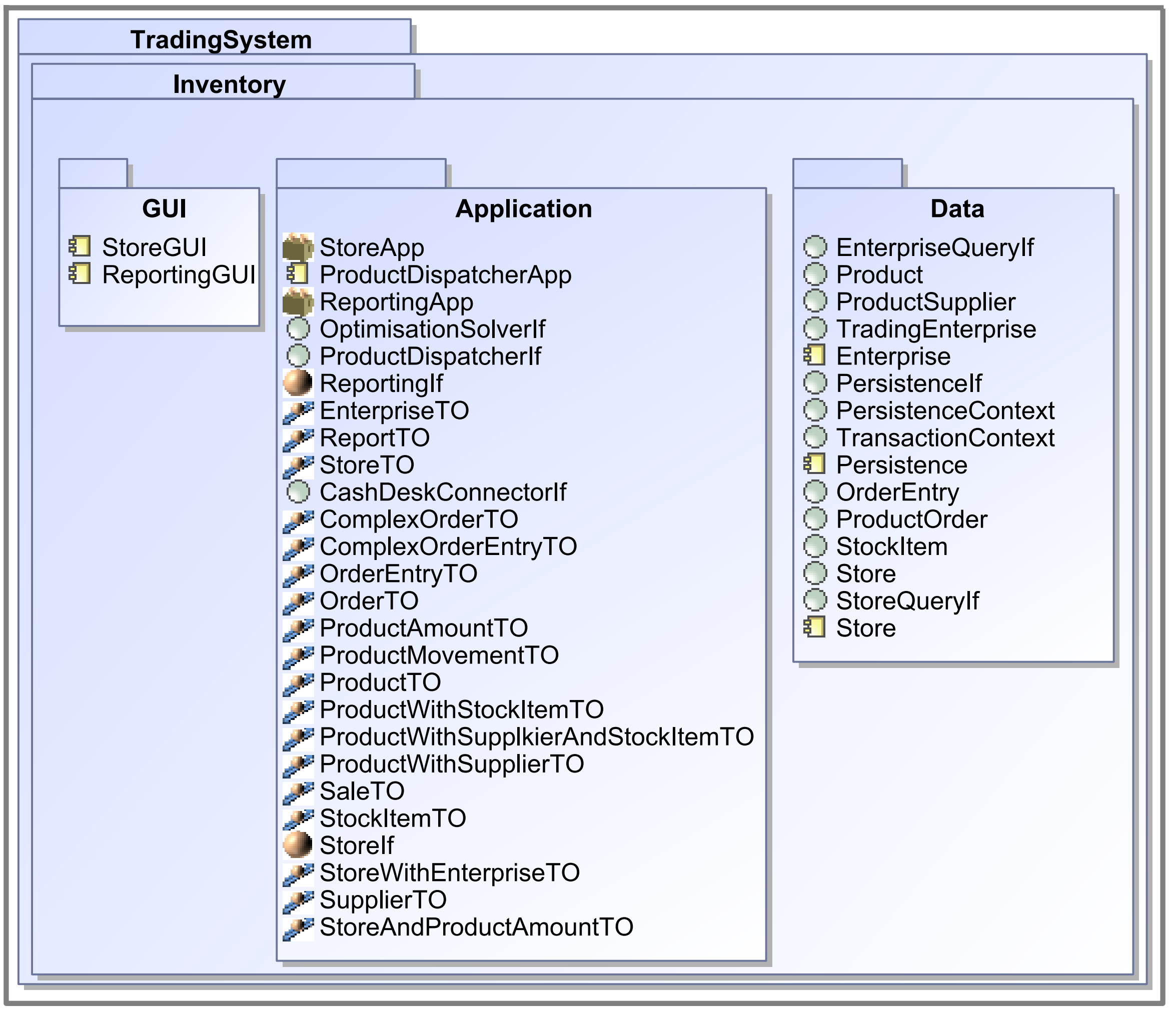


Figure C.1: Package structure of the inventory subsystem.

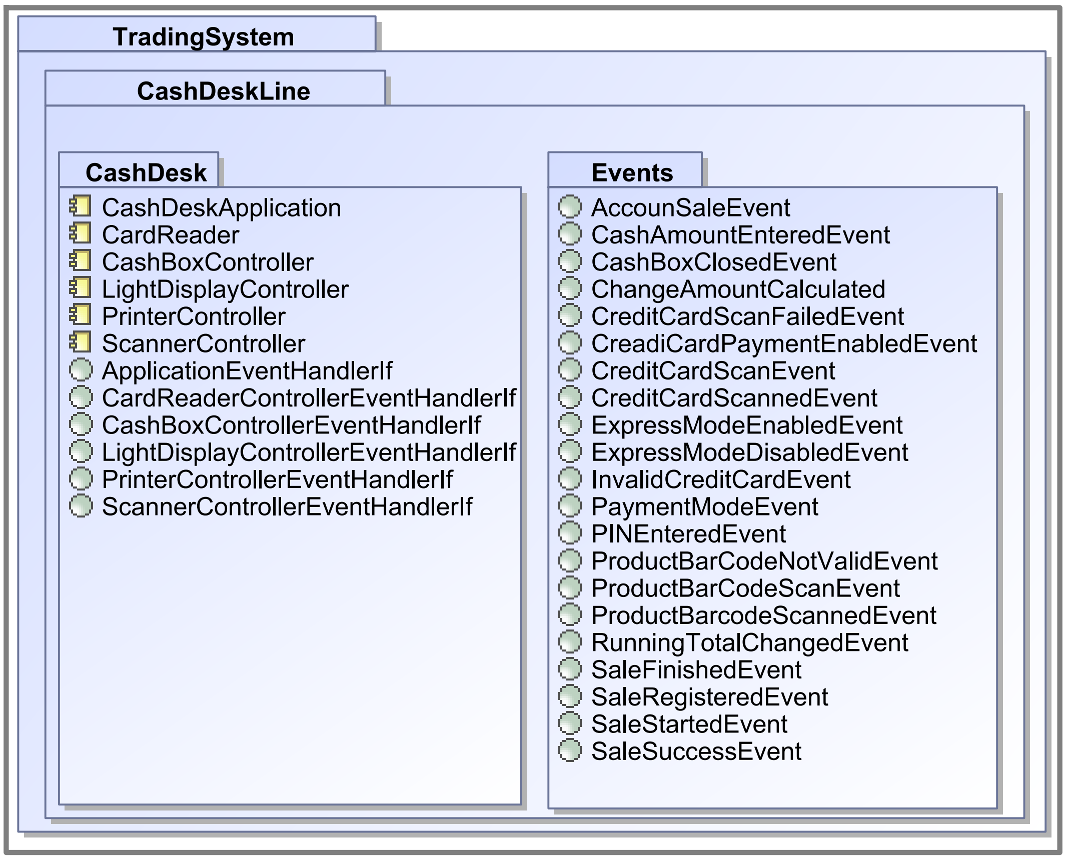


Figure C.2: Package structure of the cash desk subsystem.

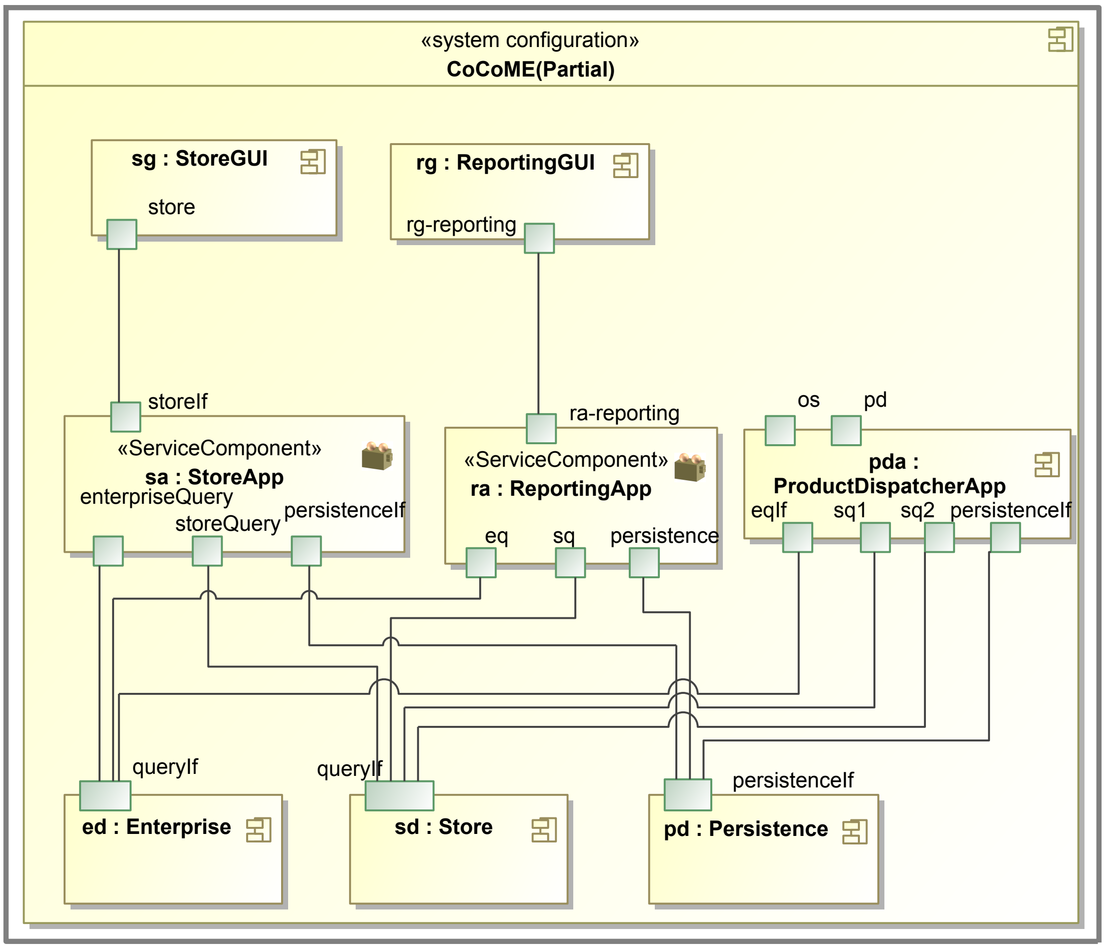


Figure C.3: Partial illustration of the CoCoME system configuration (Inventory System).

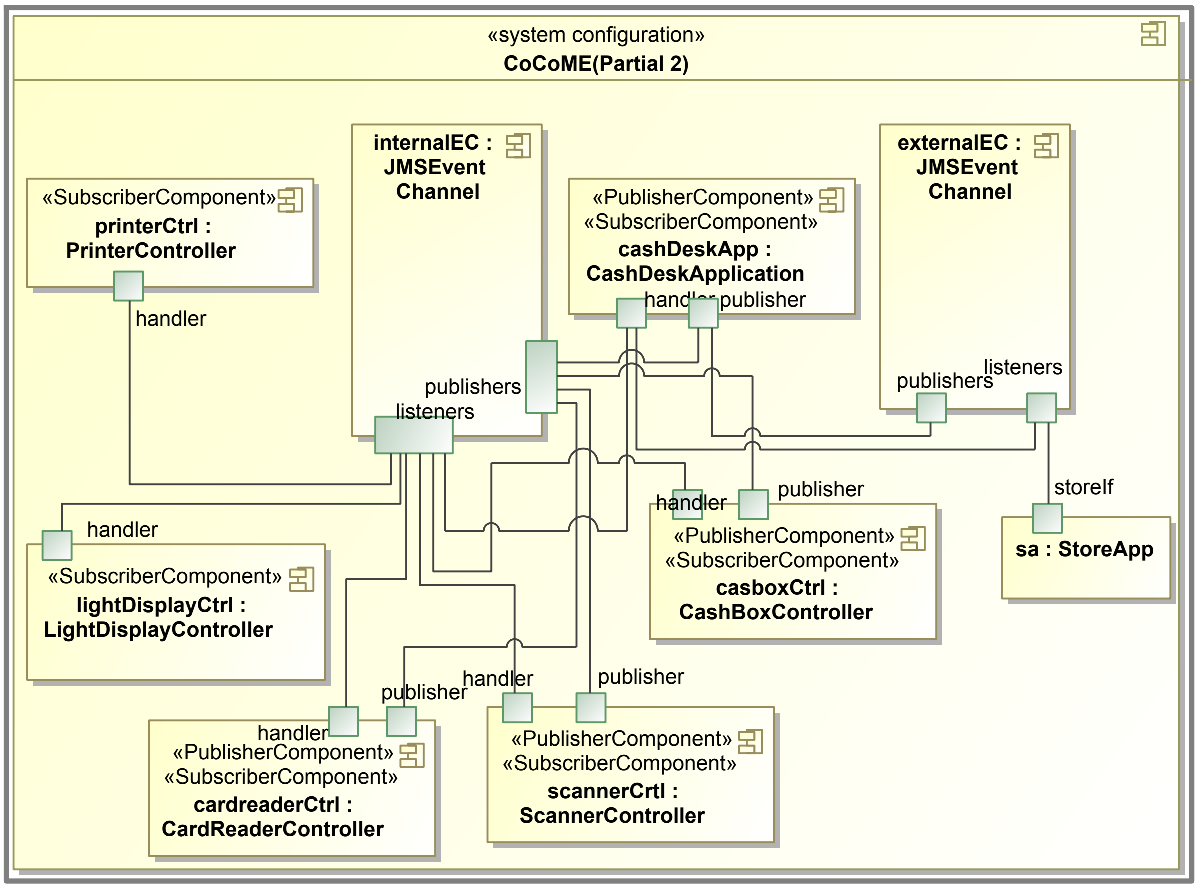
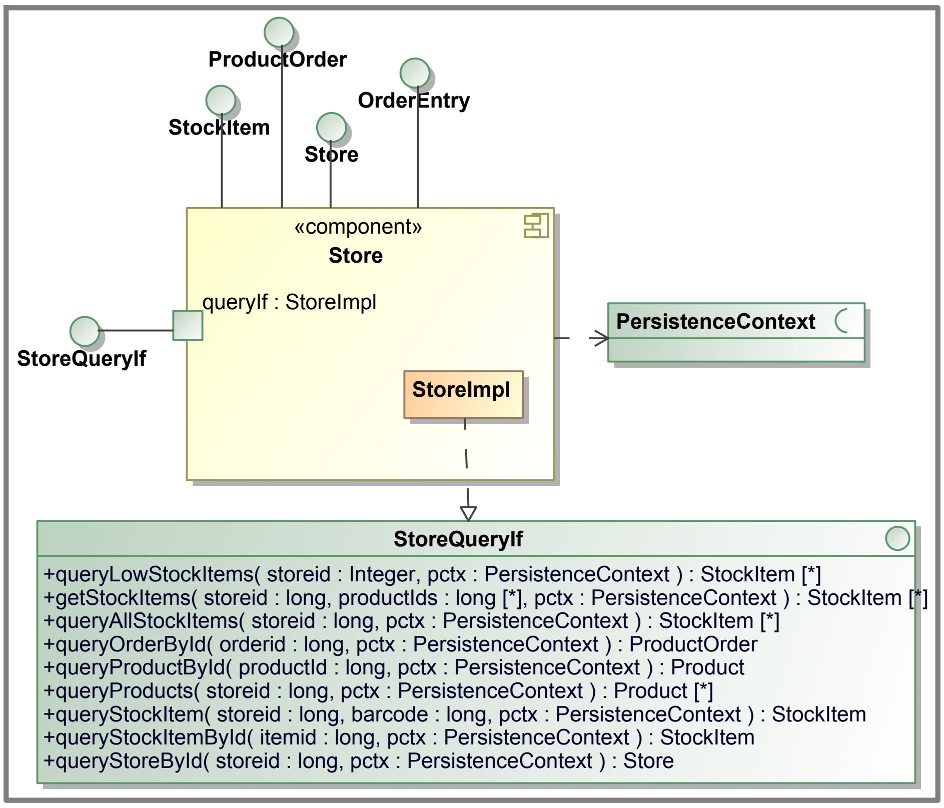
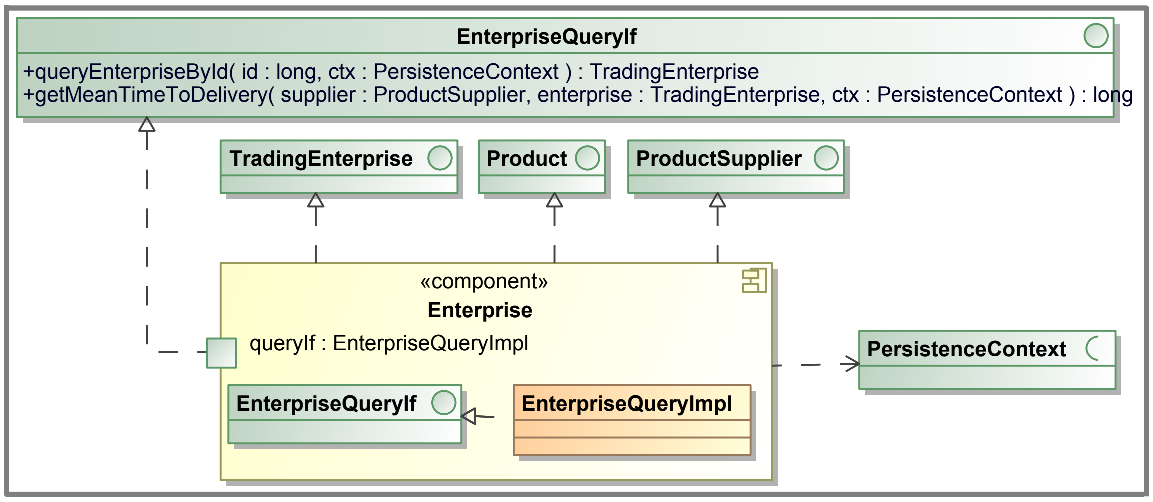


Figure C.4: Partial illustration of the CoCoME system configuration (Cashdesk System).



+queryLowStockItems(storeid : Integer, pctx : PersistenceContext) : StockItem [*]
+getStockItems(storeid : long, productIds : long [*], pctx : PersistenceContext) : StockItem [*]
+queryAllStockItems(storeid : long, pctx : PersistenceContext) : StockItem [*]
+queryOrderById(orderid : long, pctx : PersistenceContext) : ProductOrder
+queryProductById(productId : long, pctx : PersistenceContext) : Product
+queryProducts(storeid : long, pctx : PersistenceContext) : Product [*]
+queryStockItem(storeid : long, barcode : long, pctx : PersistenceContext) : StockItem
+queryStockItemById(itemid : long, pctx : PersistenceContext) : StockItem
+queryStoreById(storeid : long, pctx : PersistenceContext) : Store

Figure C.5: The Store component and related interfaces.



+queryEnterpriseById(id : long, ctx : PersistenceContext) : TradingEnterprise
+getMeanTimeToDelivery(supplier : ProductSupplier, enterprise : TradingEnterprise, ctx : PersistenceContext) : long

Figure C.6: The Enterprise component and related interfaces.

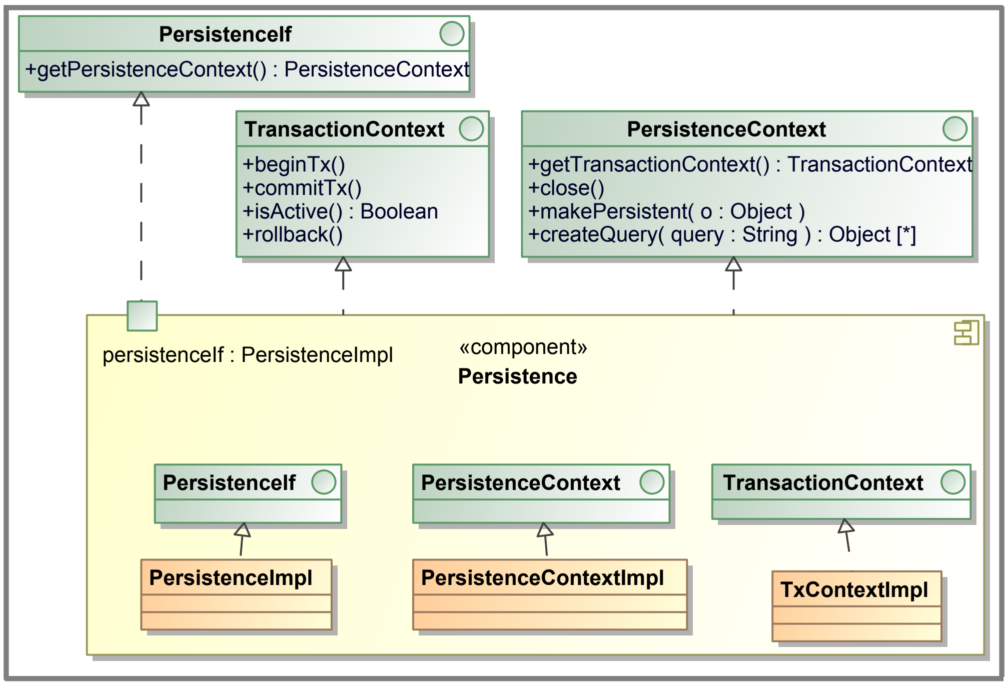


Figure C.7: The Persistence component and related interfaces.

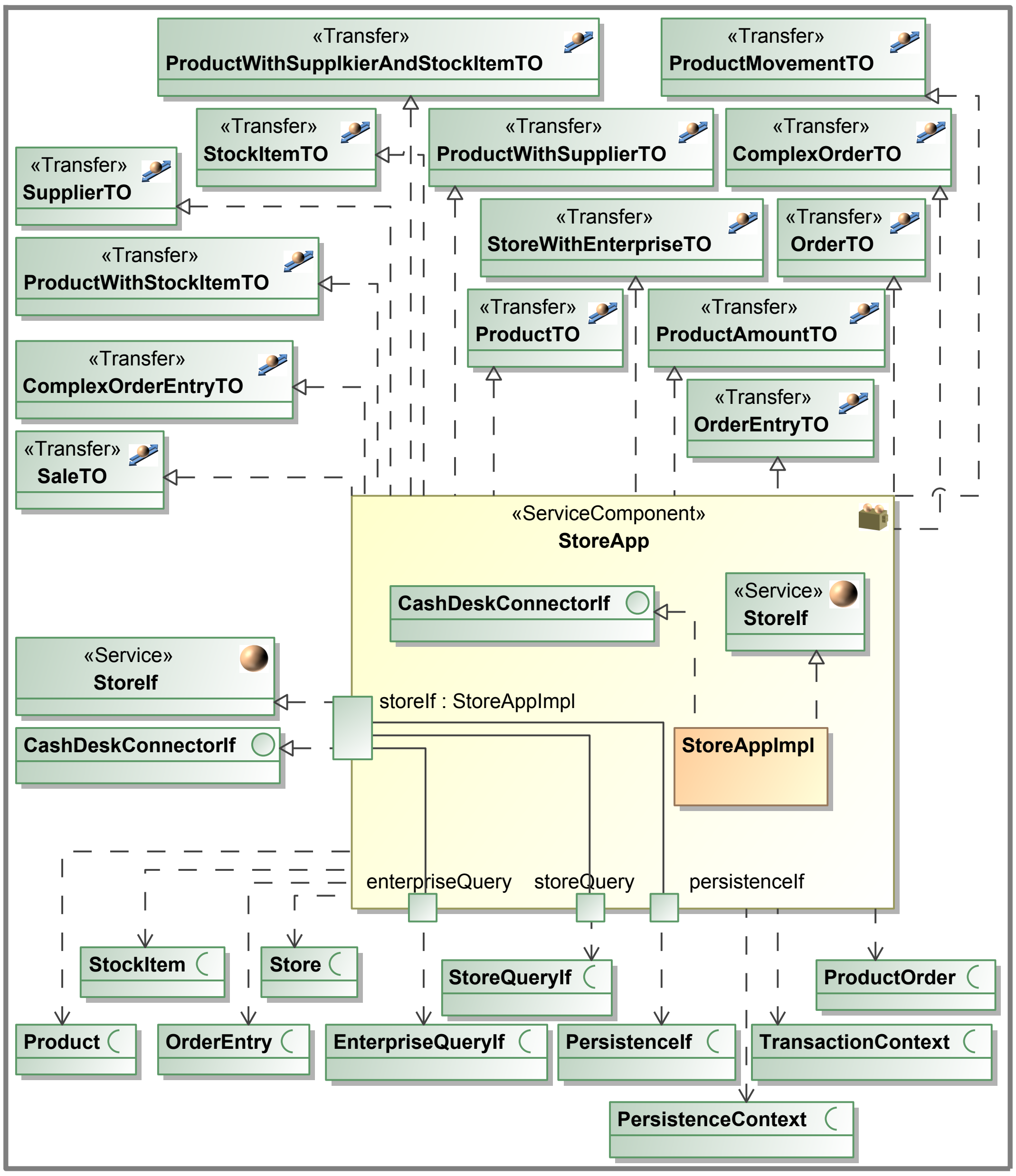


Figure C.8: The StoreApp component and related interfaces.

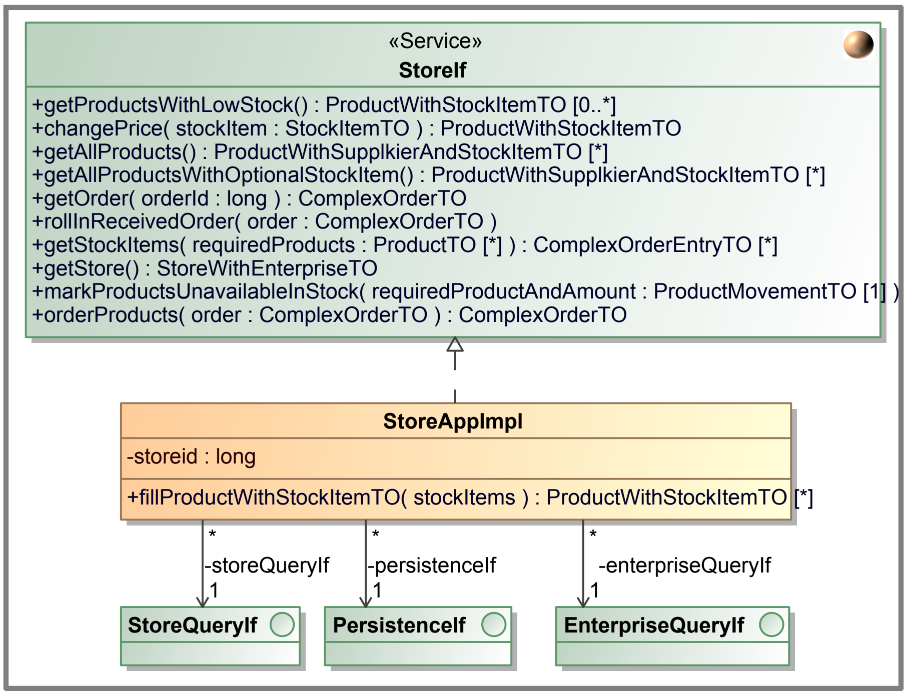

«Service»
StoreIf
+getProductsWithLowStock() : ProductWithStockItemTO [0..*]
+changePrice(stockItem : StockItemTO) : ProductWithStockItemTO
+getAllProducts() : ProductWithSupplkierAndStockItemTO [*]
+getAllProductsWithOptionalStockItem() : ProductWithSupplkierAndStockItemTO [*]
+getOrder(orderId : long) : ComplexOrderTO
+rollInReceivedOrder(order : ComplexOrderTO)
+getStockItems(requiredProducts : ProductTO [*]) : ComplexOrderEntryTO [*]
+getStore() : StoreWithEnterpriseTO
+markProductsUnavailableInStock(requiredProductAndAmount : ProductMovementTO [1])
+orderProducts(order : ComplexOrderTO) : ComplexOrderTO
StoreAppImpl
-storeid : long
+fillProductWithStockItemTO(stockItems) : ProductWithStockItemTO [*]
-storeQueryIf
-persistenceIf
-enterpriseQueryIf
StoreQueryIf
PersistenceIf
EnterpriseQueryIf

Figure C.9: Implementation details of StoreApp.

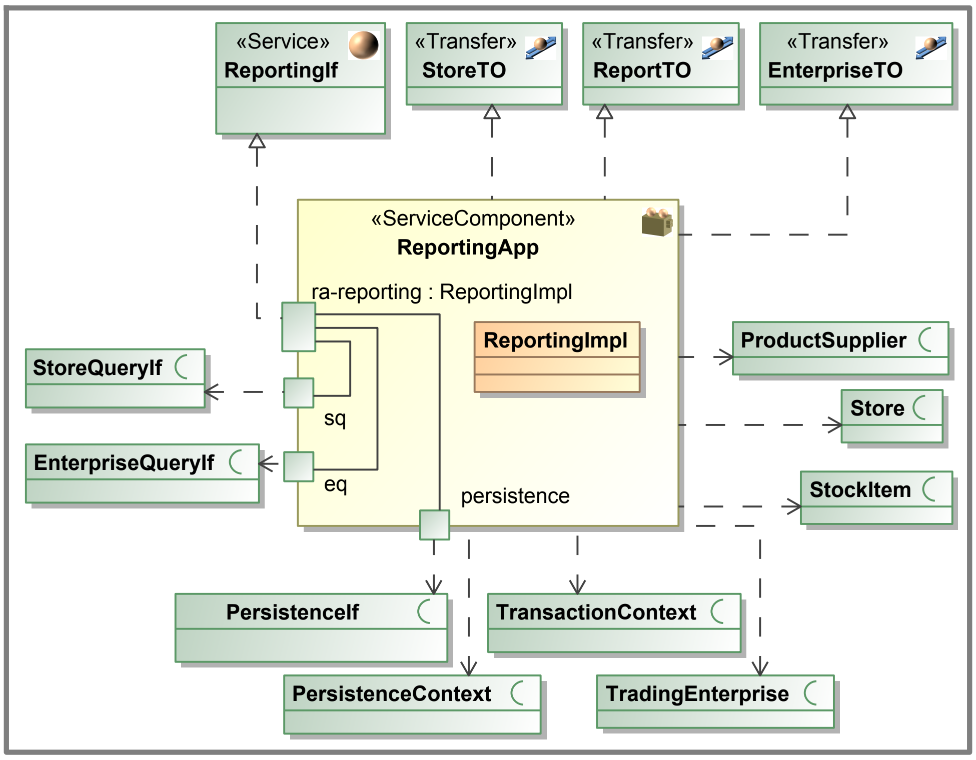


Figure C.10: The component ReportingApp and related interfaces.

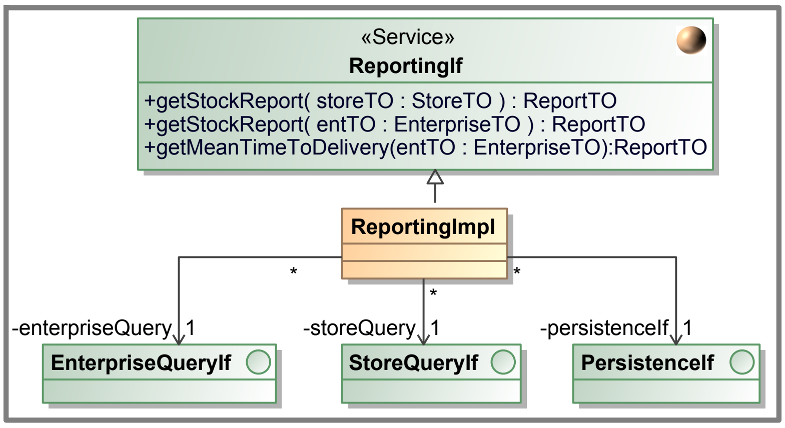

«Service»
ReportingIf
+getStockReport(storeTO : StoreTO) : ReportTO
+getStockReport(entTO : EnterpriseTO) : ReportTO
+getMeanTimeToDelivery(entTO : EnterpriseTO):ReportTO
ReportingImpl
-enterpriseQuery
-storeQuery
-persistenceIf
EnterpriseQueryIf
StoreQueryIf
PersistenceIf

Figure C.11: Implementation details of ReportingApp.

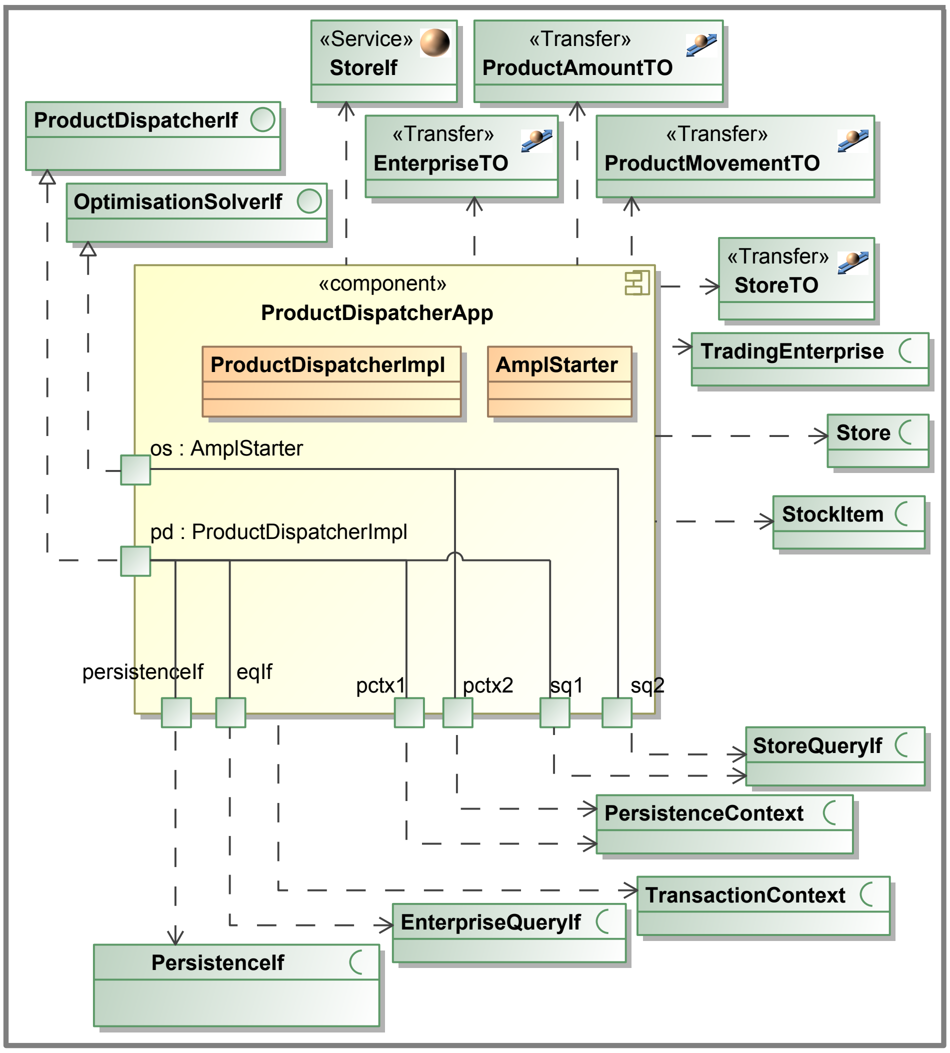
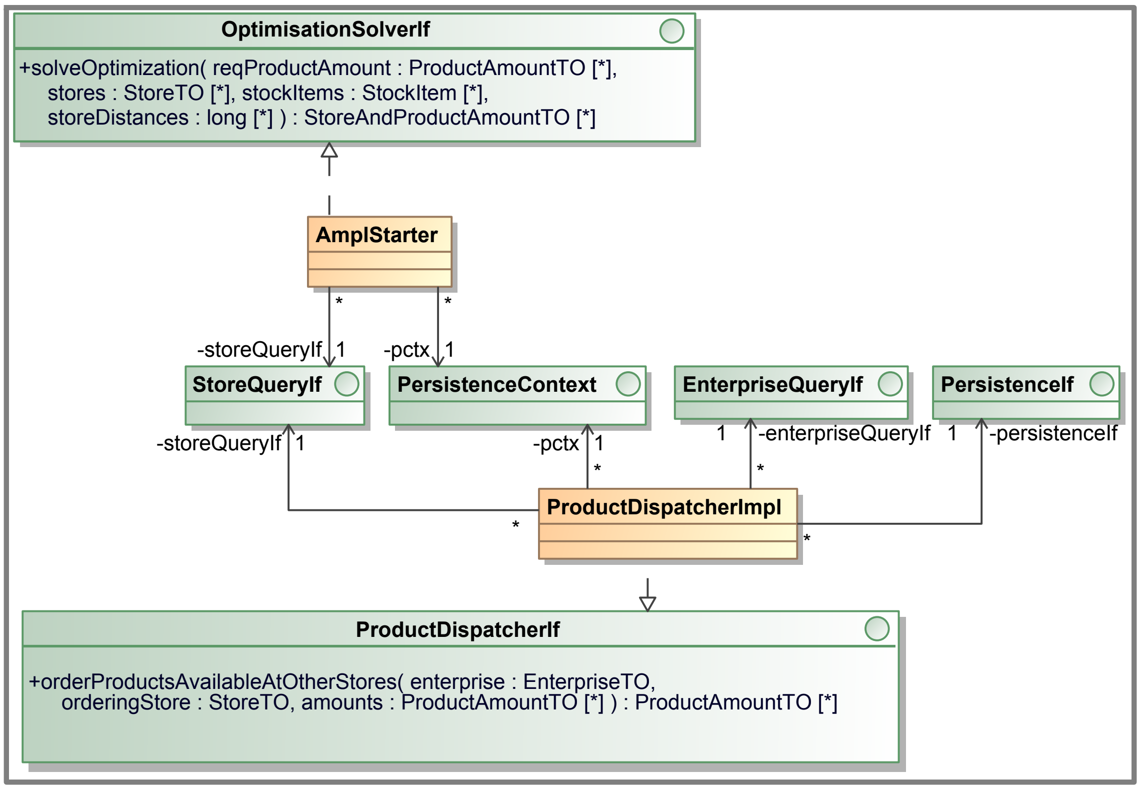


Figure C.12: The component ProductDispatcherApp and related interfaces.



OptimisationSolverIf
+solveOptimization(reqProductAmount : ProductAmountTO [*],
stores : StoreTO [*], stockItems : StockItem [*],
storeDistances : long [*]) : StoreAndProductAmountTO [*]
AmplStarter
*
*
-storeQueryIf
1
-pctx
1
StoreQueryIf
PersistenceContext
EnterpriseQueryIf
PersistenceIf
-storeQueryIf
1
-pctx
1
1
-enterpriseQueryIf
1
-persistenceIf
*
*
ProductDispatcherImpl
*
*
ProductDispatcherIf
+orderProductsAvailableAtOtherStores(enterprise : EnterpriseTO,
orderingStore : StoreTO, amounts : ProductAmountTO [*]) : ProductAmountTO [*]

Figure C.13: Implementation details of ProductDispatcherApp.

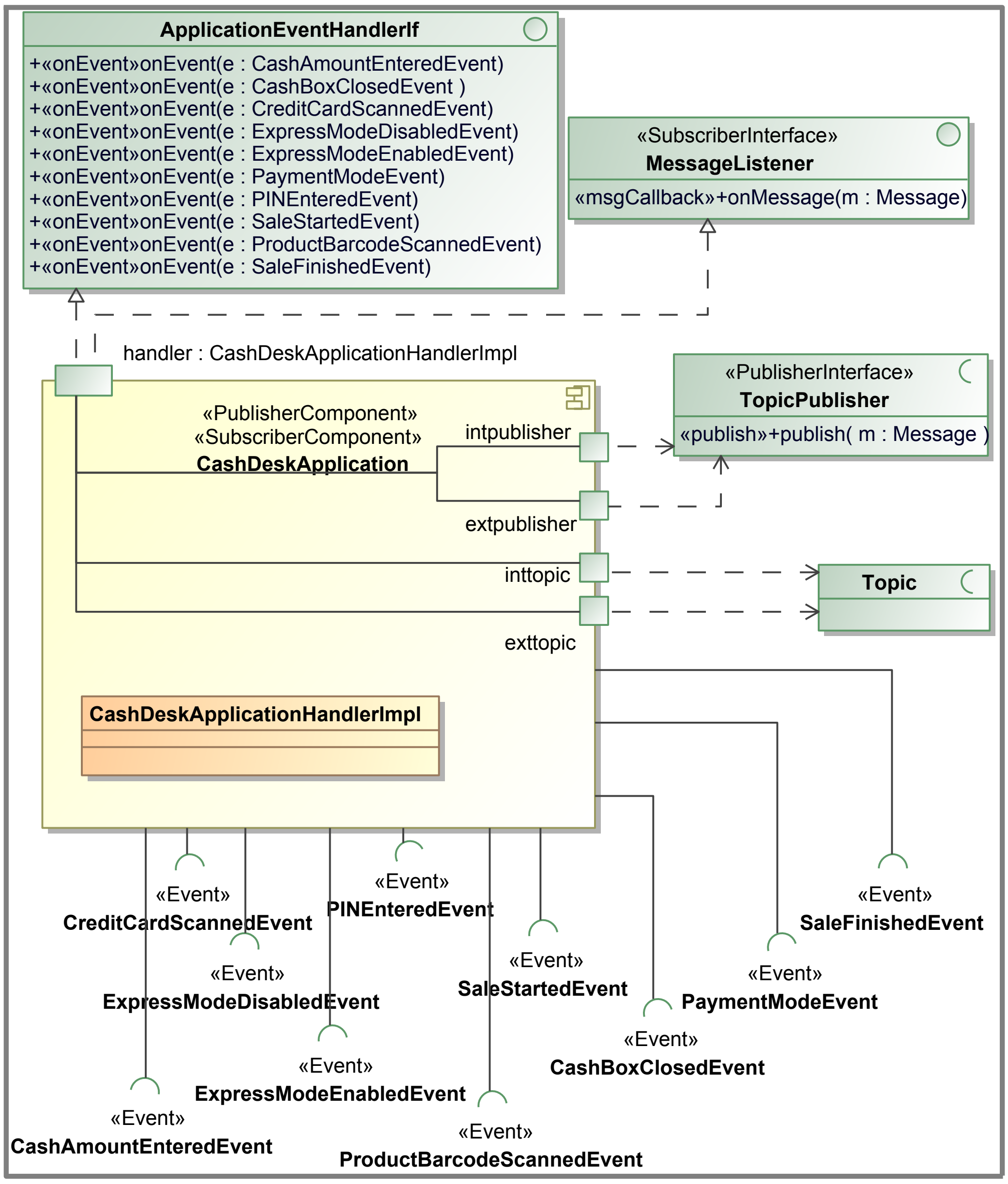

ApplicationEventHandlerIf
+«onEvent»onEvent(e : CashAmountEnteredEvent)
+«onEvent»onEvent(e : CashBoxClosedEvent)
+«onEvent»onEvent(e : CreditCardScannedEvent)
+«onEvent»onEvent(e : ExpressModeDisabledEvent)
+«onEvent»onEvent(e : ExpressModeEnabledEvent)
+«onEvent»onEvent(e : PaymentModeEvent)
+«onEvent»onEvent(e : PINEnteredEvent)
+«onEvent»onEvent(e : SaleStartedEvent)
+«onEvent»onEvent(e : ProductBarcodeScannedEvent)
+«onEvent»onEvent(e : SaleFinishedEvent)
«SubscriberInterface»
MessageListener
«msgCallback»+onMessage(m : Message)
handler : CashDeskApplicationHandlerImpl
«PublisherInterface»
TopicPublisher
«publish»+publish(m : Message)
«PublisherComponent»
«SubscriberComponent»
CashDeskApplication
intpublisher
extpublisher
inttopic
exttopic
Topic
CashDeskApplicationHandlerImpl
«Event»
CreditCardScannedEvent
«Event»
PINEnteredEvent
«Event»
SaleFinishedEvent
«Event»
ExpressModeDisabledEvent
«Event»
SaleStartedEvent
«Event»
PaymentModeEvent
«Event»
ExpressModeEnabledEvent
«Event»
CashBoxClosedEvent
«Event»
CashAmountEnteredEvent
«Event»
ProductBarcodeScannedEvent

Figure C.14: The CashDeskApplication component and related interfaces.

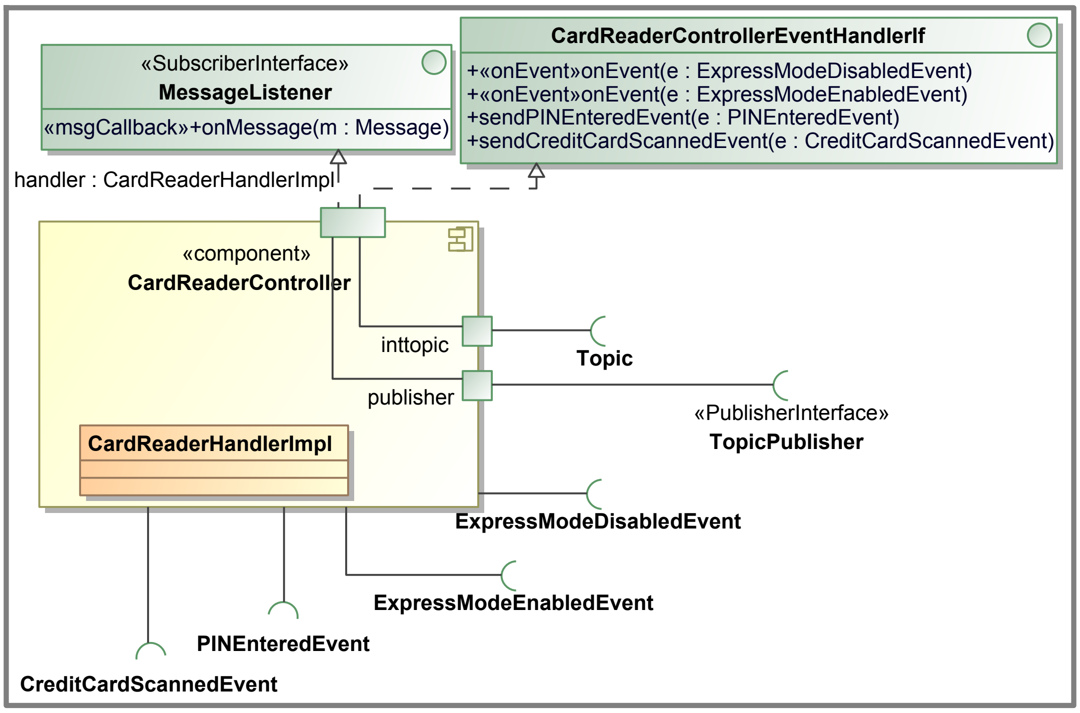

«SubscriberInterface»
MessageListener
«msgCallback»+onMessage(m : Message)
CardReaderControllerEventHandlerIf
+«onEvent»onEvent(e : ExpressModeDisabledEvent)
+«onEvent»onEvent(e : ExpressModeEnabledEvent)
+sendPINEnteredEvent(e : PINEnteredEvent)
+sendCreditCardScannedEvent(e : CreditCardScannedEvent)
handler : CardReaderHandlerImpl
«component»
CardReaderController
inttopic
publisher
Topic
«PublisherInterface»
TopicPublisher
CardReaderHandlerImpl
ExpressModeDisabledEvent
ExpressModeEnabledEvent
PINEnteredEvent
CreditCardScannedEvent

Figure C.15: The CardReader component and related interfaces.

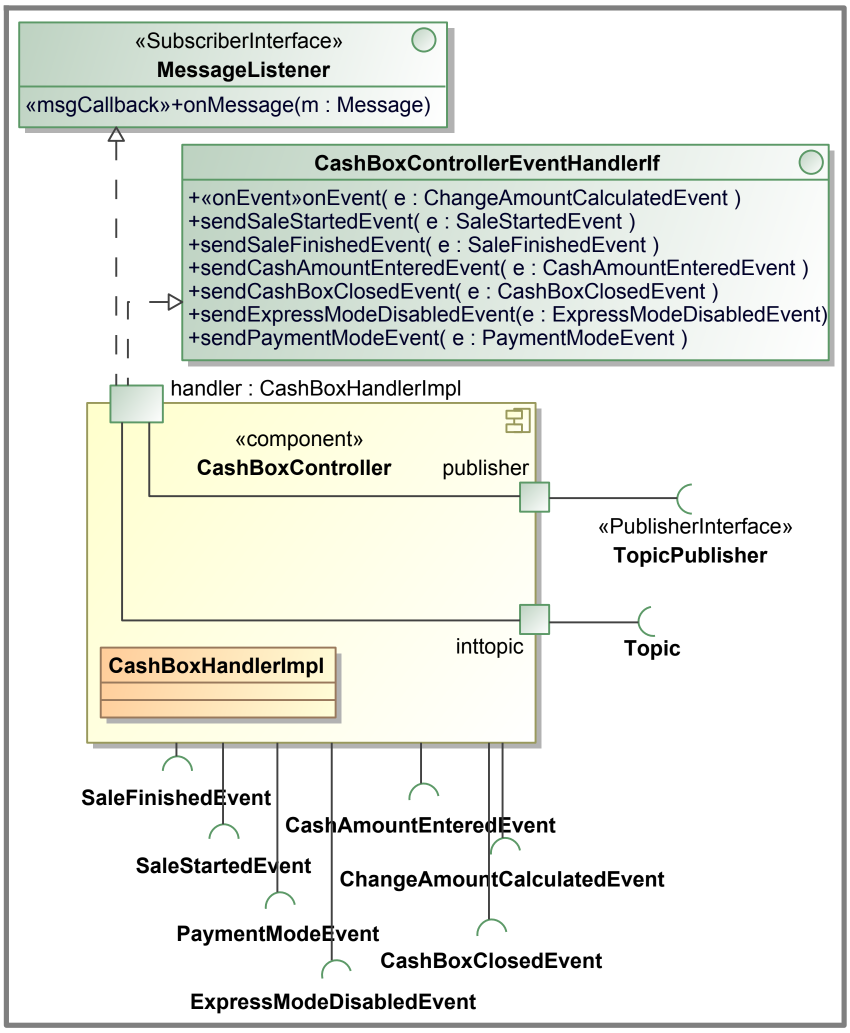

«SubscriberInterface»
MessageListener
«msgCallback»+onMessage(m : Message)
CashBoxControllerEventHandlerIf
+«onEvent»onEvent(e : ChangeAmountCalculatedEvent)
+sendSaleStartedEvent(e : SaleStartedEvent)
+sendSaleFinishedEvent(e : SaleFinishedEvent)
+sendCashAmountEnteredEvent(e : CashAmountEnteredEvent)
+sendCashBoxClosedEvent(e : CashBoxClosedEvent)
+sendExpressModeDisabledEvent(e : ExpressModeDisabledEvent)
+sendPaymentModeEvent(e : PaymentModeEvent)
handler : CashBoxHandlerImpl
«component»
CashBoxController
publisher
«PublisherInterface»
TopicPublisher
inttopic
Topic
CashBoxHandlerImpl
SaleFinishedEvent
CashAmountEnteredEvent
SaleStartedEvent
ChangeAmountCalculatedEvent
PaymentModeEvent
CashBoxClosedEvent
ExpressModeDisabledEvent

Figure C.16: The CashBox component and related interfaces.

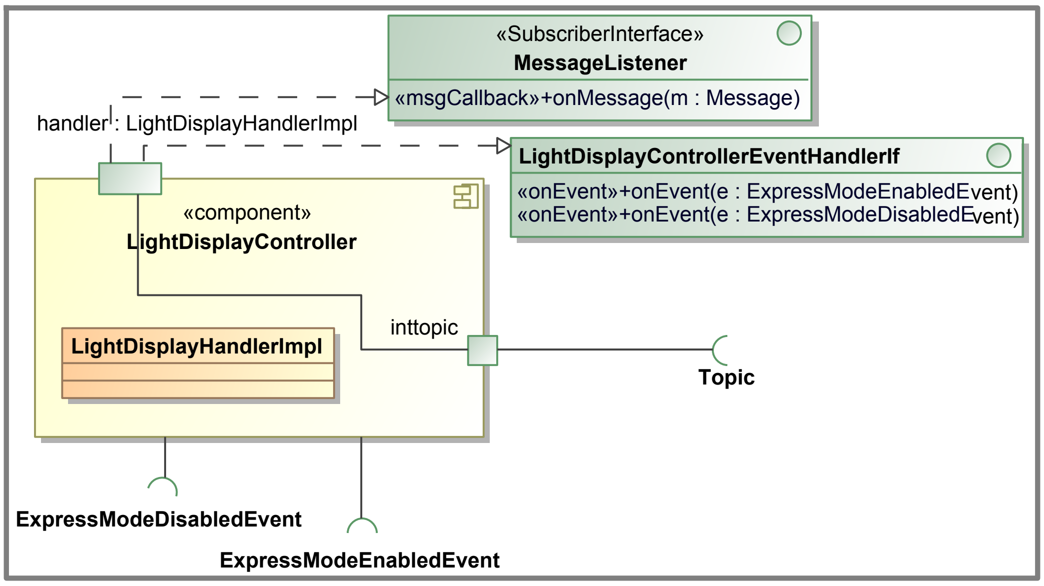

«SubscriberInterface»
MessageListener
«msgCallback»+onMessage(m : Message)
handler : LightDisplayHandlerImpl
LightDisplayControllerEventHandlerIf
«onEvent»+onEvent(e : ExpressModeEnabledEvent)
«onEvent»+onEvent(e : ExpressModeDisabledEvent)
«component»
LightDisplayController
LightDisplayHandlerImpl
inttopic
Topic
ExpressModeDisabledEvent
ExpressModeEnabledEvent

Figure C.17: The LightDisplay component and related interfaces.

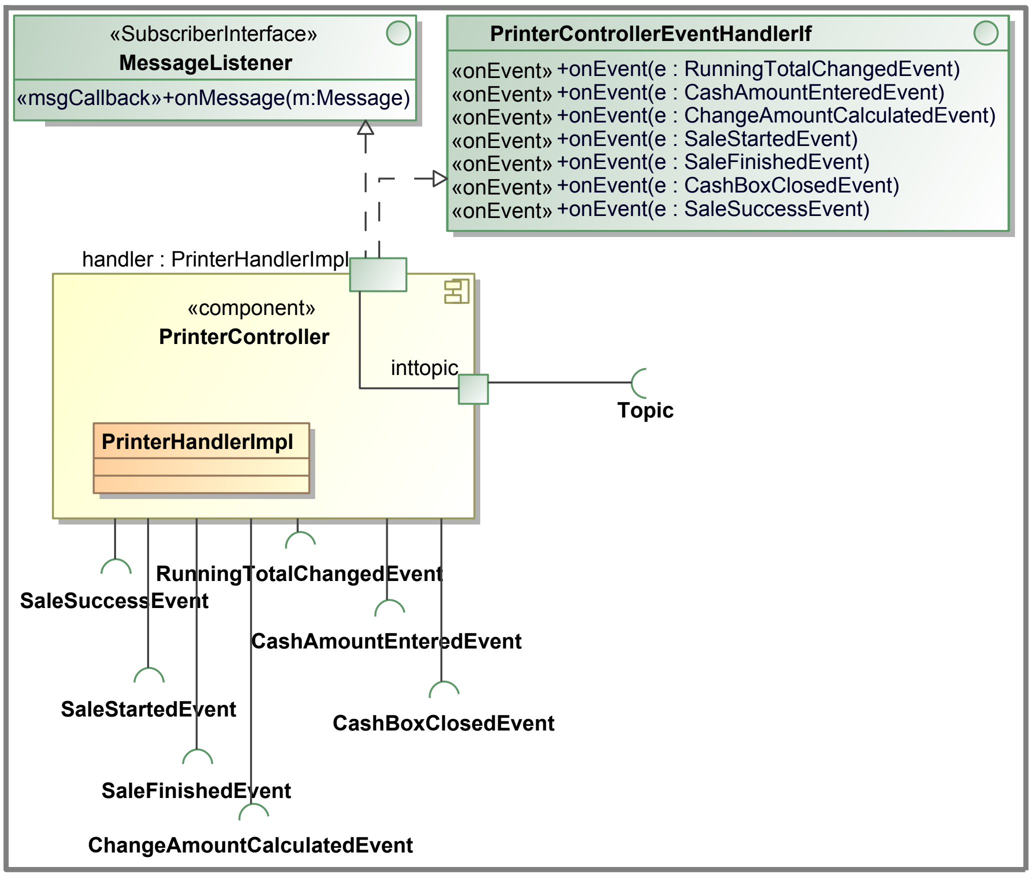

«SubscriberInterface»
MessageListener
«msgCallback»+onMessage(m:Message)
PrinterControllerEventHandlerIf
«onEvent» +onEvent(e : RunningTotalChangedEvent)
«onEvent» +onEvent(e : CashAmountEnteredEvent)
«onEvent» +onEvent(e : ChangeAmountCalculatedEvent)
«onEvent» +onEvent(e : SaleStartedEvent)
«onEvent» +onEvent(e : SaleFinishedEvent)
«onEvent» +onEvent(e : CashBoxClosedEvent)
«onEvent» +onEvent(e : SaleSuccessEvent)
handler : PrinterHandlerImpl
«component»
PrinterController
inttopic
Topic
PrinterHandlerImpl
RunningTotalChangedEvent
SaleSuccessEvent
CashAmountEnteredEvent
SaleStartedEvent
CashBoxClosedEvent
SaleFinishedEvent
ChangeAmountCalculatedEvent

Figure C.18: The Printer component and related interfaces.

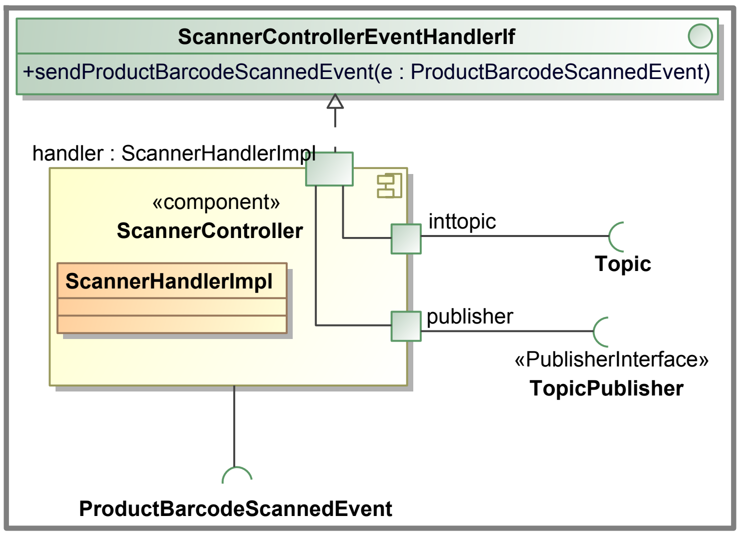

ScannerControllerEventHandlerIf
+sendProductBarcodeScannedEvent(e : ProductBarcodeScannedEvent)
handler : ScannerHandlerImpl
«component»
ScannerController
ScannerHandlerImpl
inttopic
Topic
publisher
«PublisherInterface»
TopicPublisher
ProductBarcodeScannedEvent

Figure C.19: The Scanner component and related interfaces.

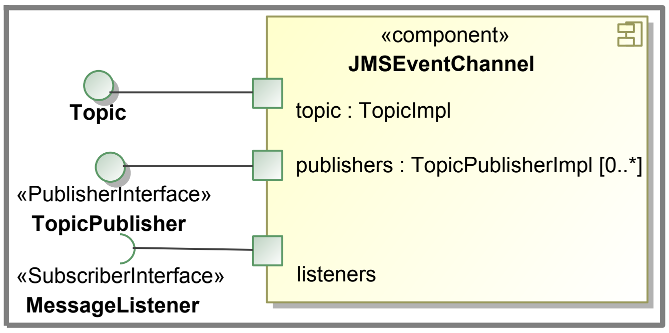


Figure C.20: The JMSEventChannel component and related interfaces.

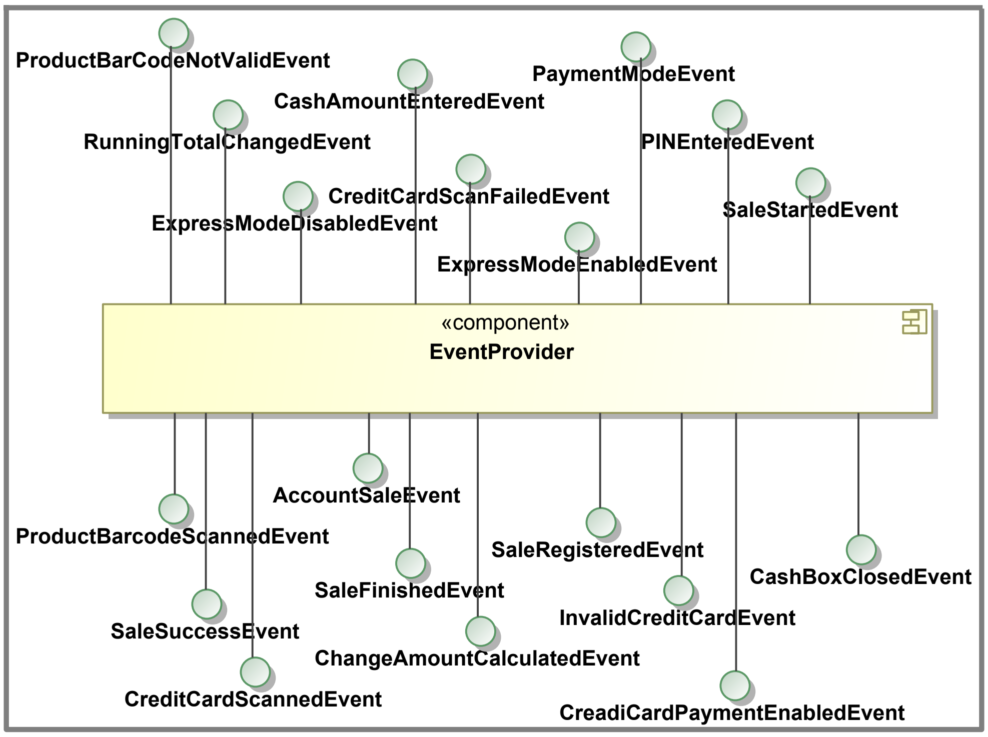


Figure C.21: Events defined in CoCoME as interfaces of a helper component EventProvider.

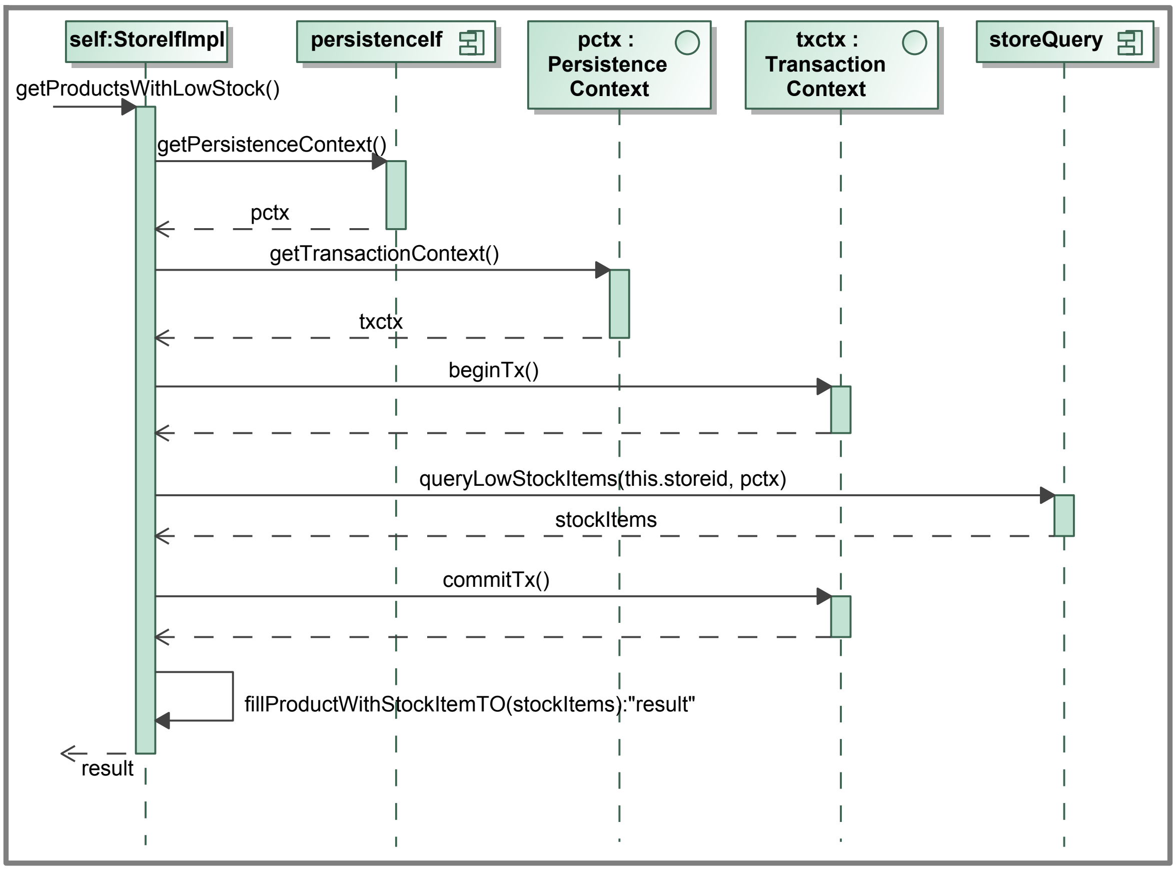

self:StoreIfImpl
persistenceIf
pctx : Persistence Context
txctx : Transaction Context
storeQuery
getProductsWithLowStock()
getPersistenceContext()
pctx
getTransactionContext()
txctx
beginTx()
queryLowStockItems(this.storeid, pctx)
stockItems
commitTx()
fillProductWithStockItemTO(stockItems):"result"
result

Figure C.22: Sequence diagram for getProductsWithLowStock implemented by component StoreApp.

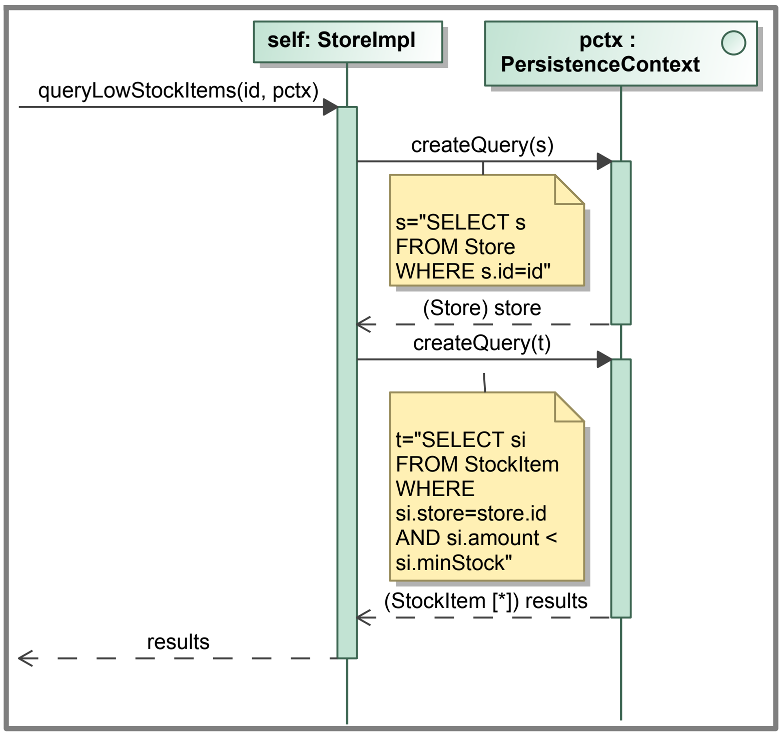

self: StoreImpl
pctx : PersistenceContext
queryLowStockItems(id, pctx)
createQuery(s)
s="SELECT s FROM Store WHERE s.id=id"
(Store) store
createQuery(t)
t="SELECT si FROM StockItem WHERE si.store=store.id AND si.amount < si.minStock"
(StockItem [*]) results
results

Figure C.23: Sequence diagram for queryLowStockItems implemented by component Store.

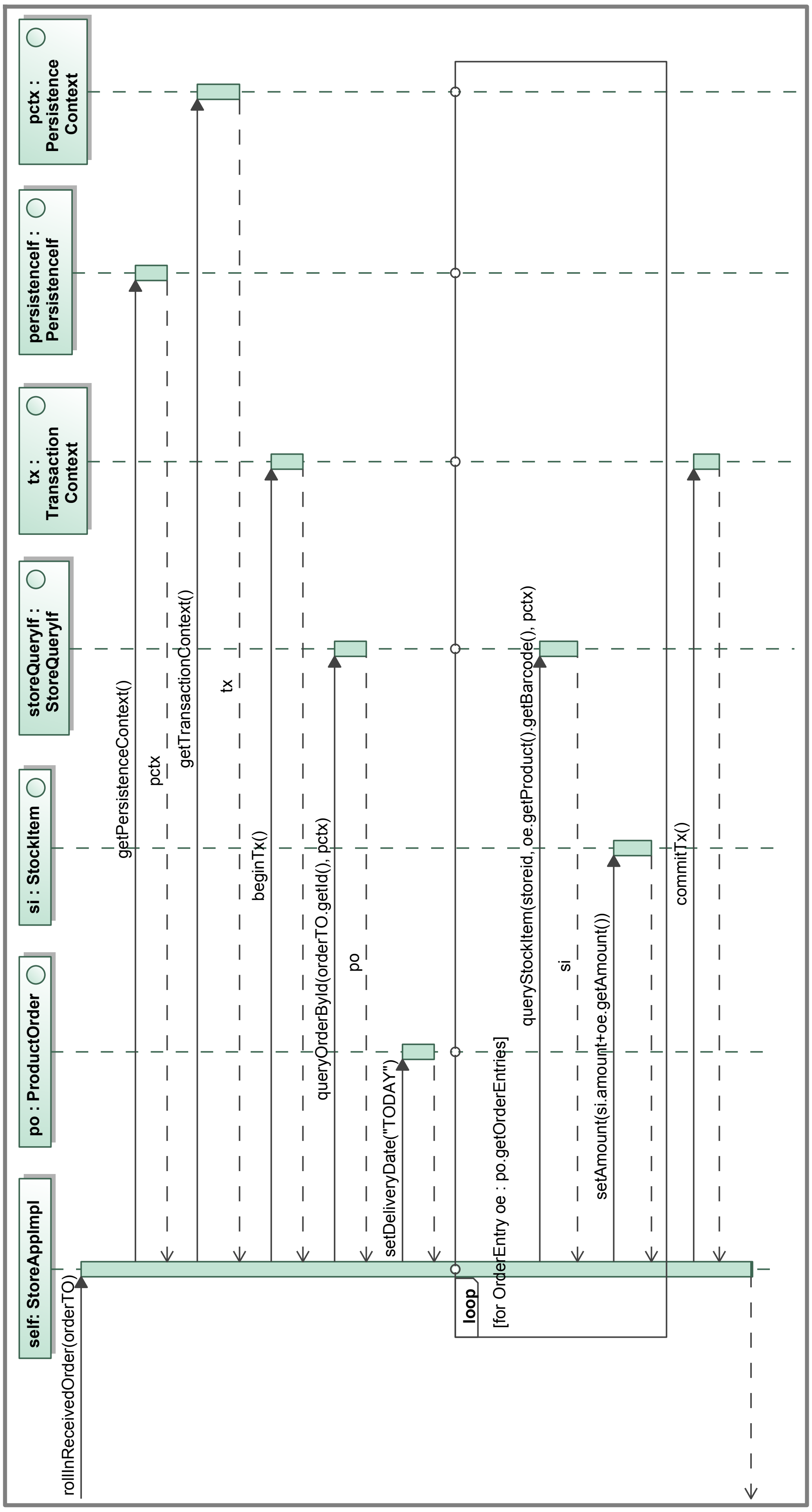

self: StoreAppImpl
po : ProductOrder
si : StockItem
storeQueryIf : StoreQueryIf
tx : TransactionContext
persistenceIf : PersistenceIf
pctx : PersistenceContext
rollInReceivedOrder(orderTO)
getPersistenceContext()
pctx
getTransactionContext()
tx
beginTx()
queryOrderById(orderTO.getId(), pctx)
po
setDeliveryDate("TODAY")
loop
[for OrderEntry oe : po.getOrderEntries]
queryStockItem(storeid, oe.getProduct().getBarcode(), pctx)
si
setAmount(si.amount+oe.getAmount())
commitTx()

Figure C.24: Sequence diagram for rollInReceivedOrder implemented by component StoreApp.

Bibliography

[AAC07] Marwan Abi-Antoun, Jonathan Aldrich, and Wesley Coelho. A case study in re-engineering to enforce architectural control flow and data sharing. *Journal of Systems and Software*, 80(2):240 – 264, 2007.

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, page 187–197, New York, USA, 2002. ACM.

[AHKR08] André Appel, Sebastian Herold, Holger Klus, and Andreas Rausch. Modelling the CoCoME with DisCComp. In *The Common Component Modeling Example: Comparing Software Component Models*, pages 267–296. Springer Berlin / Heidelberg, 2008.

[All70] Frances E Allen. Control flow analysis. In *ACM SIGPLAN Notices*, volume 5, page 1–19, New York, USA, July 1970. ACM.

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[ATL] ATL Documentation. http://www.eclipse.org/atl/documentation/. Accessed on 4th May 2011.

[Bac97] David Francis Bacon. *Fast and Effective optimization of Statically Typed Object-Oriented Languages*. PhD thesis, University of California, Berkeley, 1997.

[Bal99] Heide Balzert. *Lehrbuch der Objektmodellierung. Analyse und Entwurf.* Spektrum Akademischer Verlag, 1st edition, 1999. In German.

[Bal00] Helmut Balzert. *Lehrbuch der Software-Technik - Software-Entwicklung*. Spektrum-Akademischer Verlag, 2nd edition, 2000. In German.

[BBS03] Marcel Bennicke, Walter Bischofberger, and Frank Simon. Eclipse auf dem Prüfstand: Eine Fallstudie zur statischen Programmanalyse. *OBJEKTspektrum*, 05:22–28, 2003.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman, Amsterdam, 2nd edition, 2003.

[BCM+10] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2nd edition, 2010.

[BE08] Peter Braun and Ronny Eckhaus. Experiences on model-driven software development for mobile applications. In *Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, page 490–493, Washington, DC, USA, 2008. IEEE Computer Society.

[BGB10] Sami Beydeda, Volker Gruhn, and Matthias Book. *Model-Driven Software Development*. Springer Berlin / Heidelberg, 1st edition, 2010.

[BHLW07] Simon Becker, Sebastian Herold, Sebastian Lohmann, and Bernhard Westfechtel. A graph-based algorithm for consistency maintenance in incremental and interactive integration tools. *Software and Systems Modeling*, 6(3):287–315, 2007.

[BK06] Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning, revised edition, 2006.

[BKL04] Walter Bischofberger, Jan Kühl, and Silvio Löffler. Sotograph — a pragmatic approach to source code architecture conformance checking. In Flavio Oquendo, Brian Warboys, and Ron Morrison, editors, *Software Architecture*, volume 3047 of *Lecture Notes in Computer Science*, pages 1–9. Springer Berlin / Heidelberg, 2004.

[BLE10] Lionel C. Briand, Yvan Labiche, and Maged Elaasar. Specification and detection of modeling patterns: an approach based on QVT. Technical Report TR-SCE-10-02, Carleton University, Ottawa, 2010.

[BLW05] Paul Baker, Shiou Loh, and Frank Weil. Model-Driven engineering in a large industrial context - motorola case study. In Lionel Briand and Clay Williams, editors, *Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 476–491. Springer Berlin / Heidelberg, 2005.

[BM06] Bill Burke and Richard Monson-Haefel. *Enterprise JavaBeans 3.0*. O'Reilly, 2006.

[BMI08] BMI Bundesministerium des Innern (Federal Ministry of the Interior). V-Model XT HTML documentation. Technical report, 2008.

[BMRS96] Frank Buschmann, Regine Meunier, Hans Rohnert, and Peter Sommerlad. *A System of Patterns: Pattern-Oriented Software Architecture: Vol. 1*. John Wiley & Sons, 1st edition, 1996.

[Bro01] Tyson R. Browning. Applying the design structure matrix to system decomposition and integration problems: a review and new directions. *IEEE Transactions on Engineering Management*, 48(3):292 –306, 2001.

[BW01] Fintan Bolton and Eamon Walshe. *Pure Corba*. Sams Publishing, July 2001.

[Car56] Rudolf Carnap. *Meaning and Necessity*. University of Chicago Press, 2nd edition, 1956.

[CB05] Siobhan Clarke and Elisa Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley Longman, Amsterdam, 2005.

[CBB^{+}10] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, and Robert L. Nord. *Documenting Software Architectures: Views and Beyond*. Addison Wesley, 2nd revised edition, 2010.

[Cha01] Jorge Cham. Piled higher and deeper: Newton's three laws of graduation (second law). http://www.phdcomics.com/comics/archive.php?comicid=222, 2001. Accessed on 1st June 2011.

[CQL] Code query language specification. http://www.ndepend.com/CQL.htm. Accessed on 29th April 2011.

[CR08] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-ins*. Addison-Wesley Longman, Amsterdam, 3rd edition, 2008.

[DDHR09] Constanze Deiters, Patrick Dohrmann, Sebastian Herold, and Andreas Rausch. Rule-based architectural compliance checks for enterprise architecture management. In *Proceedings of the 13th IEEE International Conference on Enterprise Distributed Object Computing*, EDOC'09, page 158–167, Piscataway, NJ, USA, 2009. IEEE Press.

[Dep] Dependometer project website. http://sourceforge.net/projects/dependometer/. Accessed on 29th April 2011.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, page 77–101. Springer Berlin / Heidelberg, 1995.

[DH09] Constanze Deiters and Sebastian Herold. Konformität zwischen Code und Architektur - logikbasierte Überprüfung von Architekturregeln. *OBJEKTspektrum*, 05/09, 2009. In German.

[DHHJ10] Florian Deissenboeck, Lars Heinemann, Benjamin Hummel, and Elmar Juergens. Flexible architecture conformance assessment with ConQAT. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010)*, page 247–250, New York, USA, 2010. ACM.

[Dij68] Edsger W. Dijkstra. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, 11:341–346, 1968.

[DKL09] Slawomir Duszynski, Jens Knodel, and Mikael Lindvall. SAVE: software architecture visualization and evaluation. In *13th European Conference on Software Maintenance and Reengineering (CSMR '09)*, pages 323 –324, 2009.

[dMVH$^+$07] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjorn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. Keynote address: .QL for source code analysis. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, page 3–16, Washington, DC, USA, 2007. IEEE Computer Society.

[EB04] Maged Elaasar and Lionel Briand. An overview of UML consistency management. Technical Report Technical Report SCE-04-018, Department of Systems and Computer Engineering, Ottawa, 2004.

[EBL06] Maged Elaasar, Lionel Briand, and Yvan Labiche. A metamodeling approach to pattern specification. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 484–498. Springer Berlin / Heidelberg, 2006.

[EHK06] Amnon H. Eden, Yoram Hirshfeld, and Rick Kazman. Abstraction classes in software design. *IEE Proceedings - Software*, 153(4):163–182, 2006.

[EK03] Amnon H. Eden and Rick Kazman. Architecture, design, implementation. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 149–159, Portland, Oregon, 2003. IEEE Computer Society.

[EKKM08] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th International Conference on Software engineering (ICSE 2008)*, page 391–400, New York, USA, 2008. ACM.

[EN10] Ramez Elmasri and Shamkrant Navathe. *Fundamentals of Database Systems*. Prentice Hall International, 6th edition, 2010.

[Eva03] Eric J. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Longman, Amsterdam, 2003.

[FHKS09] Jan Friedrich, Ulrike Hammerschall, Marco Kuhrmann, and Marco Sihling. *Das V-Modell XT: Für Projektleiter und QS-Verantwortliche. Kompakt und übersichtlich*. Springer, Berlin, 2nd edition, 2009. In German.

[FKGS04] Robert B. France, Dae-Kyoo Kim, Sudipto Ghosh, and Eunjee Song. A UML-Based pattern specification technique. *IEEE Transactions on Software Engineering*, 30:193–206, 2004.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman, Amsterdam, 2002.

[Fre92] Gottlob Frege. Über Sinn und Bedeutung. *Zeitschrift für Philosophie und philosophische Kritik, NF 100*, pages 25–50, 1892. In German.

[GA02] Jeff Garland and Richard Anthony. *Large-Scale Software Architecture: A Practical Guide Using UML.* John Wiley & Sons, 1st edition, 2002.

[GC01] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23:685–746, November 2001.

[GeA03] Miguel Goulão and Fernando Brito e Abreu. Bridging the gap between acme and UML 2.0 for CBD. In *SAVCBS 2003 Proceedings - Specification and Verification of Component-Based Systems*, 2003.

[GF92] Michael R. Genesereth and Richard E. Fikes. *Knowledge Interchange Format Version 3.0 Reference Manual.* Stanford University, 1992.

[GFR00] Dov M. Gabbay, M. Finger, and M. Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects Volume 2.* Oxford Univ Press, 2000.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, 1995.

[GL90] Antonio Gavilanes-Franco and Francisca Lucio-Carrasco. A first order logic for partial functions. *Theoretical Computer Science*, 74(1):37–69, 1990.

[Gol89] Adele Goldberg. *SmallTalk 80: The Language.* Addison-Wesley Longman, Amsterdam, 1989.

[GP95] David Garlan and Dewayne Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4), 1995.

[Gro09] The Open Group, editor. *TOGAF Version 9.* Van Haren Publishing, 2009.

[GS94] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, 1994.

[Gup05] Samudra Gupta. *Pro Apache Log4j.* Springer Berlin / Heidelberg, 2005.

[HH06] Daqing Hou and Howard James Hoover. Using SCL to specify and check design intent in source code. *IEEE Transactions on Software Engineering*, 32:404–423, 2006.

[Hin01] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001)*, page 54–61, New York, USA, 2001. ACM.

[HKNR08] Sebastian Herold, Holger Klus, Dirk Niebuhr, and Andreas Rausch. Engineering of IT ecosystems: design of ultra-large-scale software-intensive systems. In *Proceedings of the 2nd International Workshop on Ultra-Large-Scale Software-Intensive Systems*, pages 49–52, Leipzig, Germany, 2008. ACM.

[HN99] Christine Hofmeister and Robert Nord. *Applied Software Architecture: A Practical Guide for Software Designers*. Addison-Wesley Longman, Amsterdam, 1999.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2nd edition, 2004.

[HRB+08] Sebastian Herold, Andreas Rausch, Alexander Bösl, Jan Ebell, Christian Linsmeier, and Detlef Peters. A seamless modeling approach for Service-Oriented information systems. In *Proceedings of the Fifth International Conference on Information Technology: New Generations (ITNG 2008)*, pages 438–446. IEEE Computer Society, 2008.

[IEE00] IEEE. Recommended practice for architectural description of software-intensive systems. Technical report, 2000.

[Inf06] University of Southern California Information Sciences Institute. PowerLoom documentation. http://www.isi.edu/isd/LOOM/PowerLoom/documentation/documentation.html, 2006.

[JE07] Pontus Johnson and Mathias Ekstedt. *Enterprise Architecture: Models and Analyses for Information Systems Decision Making*. Professional Pub Serv, 2007.

[JV03] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd International Conference on Aspect-OrientedSoftware Development (AOSD 2003)*, page 178–187, New York, USA, 2003. ACM.

[KA08] Ahmad Waqas Kamal and Paris Avgeriou. Modeling architectural patterns' behavior using architectural primitives. In *Proceedings of the 2nd European conference on Software Architecture (ECSA 2008)*, page 164–179. Springer Berlin / Heidelberg, 2008.

[Kaz99] Rick Kazman. A new approach to designing and analyzing Object-Oriented software architecture, 1999.

[KBS04] Dirk Krafzig, Karl Banke, and Dirk Slama. *Enterprise SOA: Service Oriented Architecture Best Practices*. Prentice Hall International, 2004.

[KBW03] Anneke Kleppe, Wim Bast, and Jos B. Warmer. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman, Amsterdam, 2003.

[KC09] Soon-Kyeong Kim and David Carrington. A formalism to describe design patterns based on role concepts. *Formal Aspects of Computing*, 21:397–420, 2009.

[KHR+03] Ludwik Kuzniarz, Zbigniew Huzar, Gianna Reggio, Jean Louis Sourrouille, and Miroslaw Staron. *Workshop on Consistency Problems in UML-based Software Development II".* 2003.

[KLMN06] Jens Knodel, Mikael Lindvall, Dirk Muthig, and Matthias Naab. Static evaluation of software architectures. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering 2006 (CSMR 2006).*, pages 285–294, 2006.

[KP07] Jens Knodel and Daniel Popescu. A comparison of static architecture compliance checking approaches. In *The Working IEEE/IFIP Conference on Software Architecture 2007 (WICSA '07)*, 2007.

[Kru95] Philippe Kruchten. Architectural Blueprints — The "4+1" view model of software architecture. *IEEE Software*, 12(6):50, 42, 1995.

[Kru03] Philippe Kruchten. *Rational Unified Process: An Introduction.* Addison-Wesley Longman, Amsterdam, 3rd edition, 2003.

[KSO06] Tahar Khammaci, Adel Smeda, and Mourad Oussalah. Mapping COSA software architecture concepts into UML 2.0. In *Proceedings of the 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering,Software Architecture and Reuse*, page 109–114, Washington, DC, USA, 2006. IEEE Computer Society.

[Lar04] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.* Prentice Hall, 3rd edition, 2004.

[LCM06] Christian F. J. Lange, Michel R. V. Chaudron, and Johan Muskens. In practice: UML software architecture and design description. *IEEE Software*, 23(2):40–46, 2006.

[LH89] Karl J. Lieberherr and Ian M. Holland. Assuring good style for Object-Oriented programs. *IEEE Software*, 6:38–48, 1989.

[LM08] Mikael Lindvall and Dirk Muthig. Bridging the software architecture gap. *IEEE Computer*, 41:98–101, 2008.

[LS06] Michael Lawley and Jim Steel. Practical declarative model transformation with Tefkat. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 139–150. Springer Berlin / Heidelberg, 2006.

[MA02] Dirk Muthig and Colin Atkinson. Model-driven product line architectures. In Gary Chastek, editor, *Software Product Lines*, volume 2379 of *Lecture Notes in Computer Science*, pages 79–90. Springer Berlin / Heidelberg, 2002.

[Mac94] Robert M MacGregor. A description classifier for the predicate calculus. In *Proceedings of the 12th National Conference on Artificial Intelligence*, AAAI '94, page 213–220, Menlo Park, USA, 1994. American Association for Artificial Intelligence.

[Mar05] Peter Marwedel. *Embedded System Design*. Kluwer Academic Publishers, 2nd edition, 2005.

[Mat10] Anders Mattsson. Automatic enforcement of architectural design rules. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010)*, page 369–372, New York, NY, USA, 2010. ACM.

[MCL04] Jeffrey Mak, Clifford Choy, and Daniel Lun. Precise modeling of design patterns in UML. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 252–261. IEEE Computer Society, 2004.

[MD08] Parastoo Mohagheghi and Vegard Dehlen. Where is the proof? - a review of experiences from applying MDE in industry. In *Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications ECMDA-FA '08*, ECMDA-FA '08, page 432–443. Springer Berlin / Heidelberg, 2008.

[Med] medini QVT website. http://projects.ikv.de/qvt/. Accessed on 4th May 2011.

[Men00] Kim Mens. *Automating Architectural Conformance Checking by Means of Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 2000.

[MHG02] David Mapelsden, John Hosking, and John Grundy. Design pattern modelling and instantiation using DPML. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications (CRPIT 2002)*, CRPIT '02, page 3–11, Darlinghurst, Australia, 2002. Australian Computer Society, Inc.

[MKB$^+$08] Mohamed N. Miladi, Mohamed H. Kacem, Achraf Boukhris, Mohamed Jmaiel, and Khalil Drira. A UML rule-based approach for describing and checking dynamic software architectures. In *IEEE/ACSInternational Conference on Computer Systems and Applications (AICCSA 2008)*, pages 1107–1114, 2008.

[ML05] Jeff McAffer and Jean-Michel Lemieux. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*. Addison-Wesley Longman, Amsterdam, 2005.

[MMM05] Sam Malek, Marija Mikic-Rakic, and Nenad Medvidovic. A Style-Aware architectural middleware for Resource-Constrained, distributed systems. *IEEE Transactions on Software Engineering*, 31:256–272, 2005.

[MNS01] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364 –380, 2001.

[MRA05] Anthony MacDonald, Danny Russell, and Brenton Atchison. Model-Driven development within a legacy system: An industry experience report. *Australian Software Engineering Conference*, 0:14–22, 2005.

[MRC09] Richard Monson-Haefel, Mark Richards, and David A. Chappell. *Java Message Service*. O'Reilly Media, 2nd edition, 2009.

[MS03] David G. Messerschmitt and Clemens Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. The MIT Press, 2003.

[MSD06] Tom Mens, Ragnhild Van Der Straeten, and Maja D'Hondt. Detecting and resolving model inconsistencies using transformation dependency analysis. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems (MoDELS 2006)*, volume 4199 of *Lecture Notes in Computer Science*, pages 200–214. Springer Berlin / Heidelberg, 2006.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[NAF07] NATO Consultation, Command and Control Board. *NATO Architecture Framework Version 3*, 2007. Accessed on 21st April 2011.

[NCEF02] Christian Nentwich, Licia Capra, Wolfgang Emmerich, and Anthony Finkelstein. xlinkit: a consistency checking and smart link generation service. *ACM Transactions on Internet Technology*, 2:151–185, May 2002.

[NDe] NDepend website. http://www.ndepend.com/Default.aspx. Accessed on 29th April 2011.

[Nie10] Dirk Niebuhr. *Dependable Dynamic Adaptive Systems — Approach, Model, and Infrastructure*. PhD thesis, Clausthal University of Technology, 2010.

[Nor06] Linda Northrop. *Ultra-large-scale systems: the software challenge of the future*. Software Engineering Institute Carnegie Mellon University, Pittsburgh, 2006.

[NR69] Peter Naur and Brian Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee*. Scientific Affairs Division, NATO, 1969.

[Obj03] Object Management Group. MDA guide version 1.0.1. Technical report, OMG (Object Management Group), 2003.

[Obj04] Object Management Group. UML profile for enterprise distributed object computing. Technical report, OMG (Object Management Group), 2004.

[Obj06] Object Management Group. Meta object facility (MOF) core specification version 2.0. Technical report, Object Management Group (OMG), 2006.

[Obj07] Object Management Group. XML metadata interchange (XMI) version 2.1.1. Technical report, Object Management Group (OMG), 2007.

[Obj08] Object Management Group. UML profile for CORBA and CORBA components specification. Technical report, Object Management Group (OMG), 2008.

[Obj10a] Object Management Group. Object constraint language version 2.2. Technical report, OMG (Object Management Group), 2010.

[Obj10b] Object Management Group. UML superstructure specification version 2.3. Technical report, Object Management Group (OMG), 2010.

[Obj11] Object Management Group. Meta object facility (MOF) 2.0 Query/View/Transformation specification. Technical report, OMG (Object Management Group), 2011.

[OFS03] Carol O'Rourke, Neal Fishman, and Warren Selkow. *Enterprise Architecture. Using the Zachman Framework*. Cengage Learning Services, 2003.

[Oqu06] Flavio Oquendo. Formally modelling software architectures with the UML 2.0 profile for π-ADL. *ACM SIGSOFT Software Engineering Notes*, 31:1–13, January 2006. ACM ID: 1108773.

[OSG03] OSGi Alliance. *OSGi Service Platform: The OSGi Alliance*. IOS Press, 2003.

[Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

[Par74] David L. Parnas. On a 'Buzzword': hierarchical structure. In *IFIP Congress*, pages 336–339, 1974.

[Ple96] John B. Plevyak. *Optimization of Object-Oriented and Concurrent Programs*. PhD thesis, University of Illinois at Urbana Champaign, 1996.

[PTV$^+$10] Leonardo Passos, Ricardo Terra, Marco T. Valente, Renato Diniz, and Nabor Mendonçanda. Static Architecture-Conformance checking: An illustrative overview. *IEEE Software*, 27(5):82 –89, 2010.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17:40–52, 1992.

[QL] .QL language reference. http://semmle.com/semmlecode/ql-language-reference/. Accessed on 29th April 2011.

[Ram04] Vishwanath Raman. Pointer analysis - a survey. *unpublished*, 2004.

[Rau04] Andreas Rausch. *Componentware - Methodik des evolutionären Architekturentwurfs*. Herbert Utz Verlag, 2004.

[RB11] Andreas Rausch and Manfred Broy. *Das V-Modell XT: Grundlagen, Erfahrungen und Werkzeuge*. Dpunkt Verlag, 2011. In German, to appear.

[RKJ04] Sunghwan Roh, Kyungrae Kim, and Taewoong Jeon. Architecture modeling language based on UML2.0. In *11th Asia-Pacific Software Engineering Conference, 2004.*, pages 663 – 669, 2004.

[RRPM08] Andreas Rausch, Ralf Reussner, Frantisek Plasil, and Raffaela Mirandola, editors. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008.

[RVP06] Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus — a tool suite for program analysis and reverse engineering. In *Ada-Europe*, volume 4006 of *Lecture Notes in Computer Science*, page 71–82. Springer Berlin / Heidelberg, 2006.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF Eclipse Modling Framework*. Addison-Wesley, 2009.

[SHR$^+$00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. *SIGPLAN Notes*, 35:264–280, 2000.

[SJSJ05] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005)*, page 167–176, New York, USA, 2005. ACM.

[Ske08] Ian Skerett. Eclipse Ganymede — 18 million lines of code delivered on schedule. Message posted at the Eclipse Zone Community Webpage. http://eclipse.dzone.com/articles/eclipse-ganymede-18-million-li, 2008. Posted on 25th June 2008. Accessed on 2nd June 2011.

[Sof] Software Engineering Institute at Carnegie Mellon University (SEI). The SEI webpage: Community software architecture definitions. Accessed on 19th April, 2011.

[Som10] Ian Sommerville. *Software Engineering*. Addison-Wesley Longman, Amsterdam, 9th edition, 2010.

[Son] SonarJ website. http://www.hello2morrow.com/products/sonarj/sonar. Accessed on 29th April 2011.

[SS94] Ehud Shapiro and Leon Sterling. *The Art of PROLOG: Advanced Programming Techniques*. MIT Press, 2nd edition, 1994.

[Sta06] Miroslaw Staron. Adopting model driven software development in industry - a case study at two companies. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems (MoDELS 2006)*, volume 4199 of *Lecture Notes in Computer Science*, pages 57–72. Springer Berlin / Heidelberg, 2006.

[Ste81] Donald V. Steward. The design structure system: a method for managing the design of complex systems. *IEEE Transactions on Software Engineering*, 28(3), 1981.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, page 32–41, New York, NY, USA, 1996. ACM. ACM ID: 237727.

[Str] Structure 101 website. http://www.headwaysoftware.com/products/?code=Structure101. Accessed on 29th April 2011.

[SVC06] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2006.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman, Amsterdam, 2nd edition, 2002.

[Tai07] Toufik Taibi. *Design Patterns Formalization Techniques.* Igi Global, 2007.

[Tex05] Mark Textor. *Über Sinn und Bedeutung von Eigennamen.* Mentis-Verlag, 1 edition, 2005.

[TMD09] Richard N. Taylor, Nenad Medvidovic, and Eric Dashofy. *Software Architecture: Foundations, Theory, and Practice.* John Wiley & Sons, 1st edition, 2009.

[TP00] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2000)*, page 281–293, New York, USA, 2000. ACM.

[TV09] Ricardo Terra and Marco Tulio Valente. A dependency constraint language to manage object-oriented software architectures. *Software Practice and Experience*, 39:1073–1094, 2009.

[TvS08] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall International, 2nd edition, 2008.

[UNKC08] M. Usman, A. Nadeem, Tai-hoon Kim, and Eun-suk Cho. A survey of consistency checking techniques for UML models. In *Advanced Software Engineering and Its Applications, 2008. ASEA 2008*, pages 57 –62, 2008.

[vdAvH04] Wil van der Aalst and Kees van Hee. *Workflow Management: Models, Methods, and Systems.* MIT Press, 2004.

[Vol98] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 1998.

[Vol06] Kris De Volder. JQuery: a generic code browser with a declarative configuration language. In Pascal Van Hentenryck, editor, *Practical Aspects of Declarative Languages*, volume 3819 of *Lecture Notes in Computer Science*, pages 88–102. Springer Berlin / Heidelberg, 2006.

[WR05] Eoin Woods and Nick Rozanski. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Longman, Amsterdam, 2005.

[Wuy01] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

[XDe] XDepend website. http://www.xdepend.com/. Accessed on 29th April 2011.

[YPMT09] Angela Yochem, Leslie Phillips, Frank Martinez, and Hugh Taylor. *Event-Driven Architecture*. Addison-Wesley Longman, Amsterdam, 1 edition, 2009.

[ZA08] Uwe Zdun and Paris Avgeriou. A catalog of architectural primitives for modeling architectural patterns. *Information and Software Technology*, 50(9-10):1003–1034, 2008.