Validation of Architectural Communication Integrity Based on Run-Time-Monitoring of Software Systems

Johannes Arthur Dohmen

March 10, 2014

Reviewers
Prof. Dr. rer. nat. Horst Lichter
Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor
M.Sc. Ana-Maria-Cristina Dragomir
I hereby affirm that the Master Thesis at hand is my own written work and that I have used no other sources and aids other than those indicated. All passages quoted from publications or paraphrased from these sources are properly cited and attributed.

The thesis was not submitted in the same or in a substantially similar version, not even partially, to another examination board and was not published elsewhere.

Aachen, March 10, 2014

(Johannes Arthur Dohmen)
Abstract

Software systems evolve during their lifetime which is known as Lehman’s first law of software evolution, “Continuing Change”. Due to this “Continuing Change” it is possible that a gap opens between the system’s architecture description, the conceptual architecture, and its realization, the concrete architecture, as both can be manipulated independently thus allowing for the risk of them “drifting apart” from each other over the lifetime of the software system. This is known as the “architectural gap” between conceptual and concrete architecture. Architectural gaps often manifest through components of the concrete architecture interacting differently from the ways stipulated by the conceptual architecture. An architectural gap violates the communication integrity of the software system. The presented thesis demonstrates an approach to detect such violations through runtime monitoring of the software system. For the realization of the detection of architectural gaps a metamodel allowing to model a software system’s architecture along with its communication rules and abilities to connect the model to the source code is provided. Furthermore a validation algorithm based on an architecture model and runtime monitoring is presented. Finally the implementation of CommunicationIntegrityChecker (CIC) as an implementation of the concept is presented. It is hoped that this approach will prove very useful for the detection of violations of the communication integrity thusly allowing for their removal as to close or at least minimize architectural gaps.
Deutsch

# Contents

1. Introduction 1  
   1.1. Motivation ....................................................... 1  
   1.2. Structure of This Thesis ....................................... 3  
   1.3. Introductory Example .......................................... 3  

2. Background 9  
   2.1. Terminology .................................................... 10  

3. Related Work 13  
   3.1. Architecture Description Languages .......................... 13  
   3.2. Connecting Architecture and Implementation .................. 15  
   3.3. Static Analysis .................................................. 19  
   3.4. Dynamic Analysis ............................................... 21  

4. Concept 25  
   4.1. Code-Level Entities ............................................. 26  
   4.2. Architectural Entities ......................................... 28  
   4.3. Include Relations (∈) .......................................... 29  
   4.4. Communication Rules (↝) ...................................... 31  
   4.5. Disallowing Communication Rules (↛) ......................... 35  
   4.6. Resolving the Ambiguity of Rules ............................. 37  
   4.7. Software Architecture Metamodel .............................. 38  
   4.8. The Need for Default Rules .................................... 38  
   4.9. Implicit Communication Rules .................................. 40  

5. Realization 41  
   5.1. CIC Class Library ............................................... 41  
   5.2. Validation ......................................................... 53  
   5.3. Tools ............................................................... 62  

6. Evaluation 65  
   6.1. MosAIC ............................................................. 65  
   6.2. Evaluation of the MosAIC Web Application ..................... 66  
   6.3. CIC ................................................................. 73  
   6.4. Summary .......................................................... 74
## List of Tables

<table>
<thead>
<tr>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1. Code-level entities</td>
<td>26</td>
</tr>
<tr>
<td>6.1. Architectural entities defined for the MosAIC web application</td>
<td>69</td>
</tr>
<tr>
<td>6.2. Rules defined for the MosAIC web application</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule sets marked with (*) were not retrieved from the architecture description but communicated by the author.</td>
<td>70</td>
</tr>
<tr>
<td>6.3. Violations after the first validation attempt</td>
<td>72</td>
</tr>
<tr>
<td>6.4. Violations after the second validation attempt</td>
<td>73</td>
</tr>
<tr>
<td>6.5. Rules defined for CIC</td>
<td>78</td>
</tr>
<tr>
<td>6.6. Rules defined for CIC (connections). Abbreviations:</td>
<td>79</td>
</tr>
<tr>
<td>m=monitoring,x=xmpp,v=validation,a=architecture</td>
<td></td>
</tr>
</tbody>
</table>
## List of Figures

1.1. Example architecture and corresponding classes ........................................ 4
1.2. Example classes in the context of the example architecture ......................... 4
1.3. Classes and exemplary execution sequence of their instances ....................... 5
1.4. Informal architecture notation ..................................................................... 6

3.1. LISA Overview [BW09] ................................................................................. 16
3.2. LISA Overview [BW09] ................................................................................. 16
3.3. Two views of the same violation [BW10] ....................................................... 17
3.4. A graphical compiler architecture [ACN02a] .................................................. 18
3.5. Component definitions in ArchJava [ACN02a] ............................................... 19
3.7. Screenshot: Structural Compliance Checking Results [DKL09] ...................... 21
3.8. The DiscoTect Architecture [YGS*04] .......................................................... 22
3.9. Kieker traces [vHRH*09] ................................................................................. 23

4.1. Code-level entity ............................................................................................. 27
4.2. Code-level entity ............................................................................................. 27
4.3. Architectural entity including... ..................................................................... 29
4.4. Entity tree with controller entity ..................................................................... 30
4.5. Communication rules ..................................................................................... 32
4.6. Communication rule between including entities ............................................. 33
4.7. Derived communication rules ....................................................................... 33
4.8. Ambiguous rules ........................................................................................... 35
4.9. Disallowing Communication rules .................................................................. 35
4.10. Example facade ............................................................................................. 36
4.11. Contradictory rules ....................................................................................... 36
4.12. Ambiguous rules .......................................................................................... 37
4.13. Metamodel of architectural information ...................................................... 39

5.1. Simplified class diagram of the CIC class library (UML) ............................. 42
5.2. Detailed UML class diagram of package rules ............................................... 43
5.3. Detailed UML class diagram of package architecture ................................... 44
5.4. Detailed UML class diagram of package monitoring .................................... 45
5.5. Detailed UML class diagram of package validation ...................................... 47
5.6. Logos ............................................................................................................. 49
5.7. Execution record pair (XMPP message) ........................................................ 50
5.8. Simplistic approach to find all matching rules (Nassi-Shneiderman diagram) 56
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.9</td>
<td>High-level View of a Rule Engine</td>
<td>58</td>
</tr>
<tr>
<td>5.10</td>
<td>CIC tasks</td>
<td>63</td>
</tr>
<tr>
<td>5.11</td>
<td>CIC tasks with Xmpp communication</td>
<td>63</td>
</tr>
<tr>
<td>6.1</td>
<td>MosAIC architecture overview</td>
<td>66</td>
</tr>
<tr>
<td>6.2</td>
<td>MosAIC architecture with the web tier</td>
<td>67</td>
</tr>
<tr>
<td>6.3</td>
<td>MosAIC architecture with the business tier</td>
<td>68</td>
</tr>
<tr>
<td>6.4</td>
<td>CIC architecture with communication rules</td>
<td>77</td>
</tr>
</tbody>
</table>
### List of Source Codes

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Import of all entities embedded by a (sub) package</td>
</tr>
<tr>
<td>5.1</td>
<td>Architecture import (XML definition)</td>
</tr>
<tr>
<td>5.2</td>
<td>Rules import (XML definition)</td>
</tr>
<tr>
<td>5.3</td>
<td>Validator.validate</td>
</tr>
<tr>
<td>5.4</td>
<td>SimpleValidationStrategy.findAllMatchingRules</td>
</tr>
<tr>
<td>5.5</td>
<td>Schema of drools rules</td>
</tr>
<tr>
<td>5.6</td>
<td>A simple underage check [JB14a]</td>
</tr>
<tr>
<td>5.7</td>
<td>DroolsValidationStrategy derive_rules.drl</td>
</tr>
<tr>
<td>5.8</td>
<td>DroolsValidationStrategy remove_ambiguous_rules.drl</td>
</tr>
<tr>
<td>5.9</td>
<td>DroolsValidationStrategy validation.drl</td>
</tr>
<tr>
<td>6.1</td>
<td>Extract of the code-level entity definition</td>
</tr>
<tr>
<td>A.1</td>
<td>architecture.xsd</td>
</tr>
<tr>
<td>A.2</td>
<td>rules.xsd</td>
</tr>
</tbody>
</table>
1. Introduction

"A little inaccuracy saves a world of explanation".

C.E. Ayers

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Structure of This Thesis</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Introductory Example</td>
<td>3</td>
</tr>
</tbody>
</table>

1.1. Motivation

As a result of ever-enlarging software systems designing a software system has become to be acknowledged as a separate activity of software development since the 1970s [WP92]. As Grady Booch stated: “You don’t need architecture to build a dog kennel, but you’d better have some for a skyscraper” [FCA+10], therefore it has been considered that software systems exceeding a certain size can only be successfully realized when designed in advance. The term software architecture is similar to the term software design but until now there has not been a clear and broadly accepted distinction between both. Some authors use both terms synonymously while others emphasize a difference in the level of abstraction. Yet other authors use the term design in the sense of the activity and the term architecture as the result of that activity [LL10]. The presented thesis follows the latter approach, although it should be noted that the architecture of a software system is of much more importance to this thesis than the activity creating it. Grady Booch’s quote already introduced architecture in the sense of making buildings as a common analogy of software architecture. That analogy has—as all analogies do—its shortcomings but it does underline the fact that software systems require to be designed before construction and manifested blueprints, plans and designs to guide the construction as much as buildings do. Those blueprints, plans and designs are commonly referred to as the architecture description of the software system. There is no guarantee that all plans and blueprints were followed when implementing a software system similar to the process of construction. Quite the contrary, it seems likelier that the architecture description is not followed—and thus being violated—more often than in construction work. Not following the intended architecture while building a house will lead to unwanted results that may range from minor flaws to catastrophic events. The same is true for software systems as implementing them ignorant of the intended
1. Introduction

architecture may lead to shortcomings in one or more of the qualities it shall provide or may even render it completely unusable. Even if the initial implementation of a software system originally matches its architecture description later changes may alter the system in unintended ways. These modifications require updating the architecture description according to the changes. If the architecture description is not updated a gap between the architecture description and the implementation opens which diminishes the value of the architecture description in the same way as original plans of a building lose more and more of their value on each alteration of the building. This thesis presents an approach to identify such gaps between the architecture description and implementation of a software system.

The software architecture description is an artifact created during the design phase. Most if not all software development processes contain such a design phase. In the waterfall model the design phase is a separate phase, while in newer iterative development processes it is a recurring phase in the process iterations. Despite its importance the software architecture description of a system is often missing, incomplete, (partly) invalid or outdated. The reconstruction of (partly) missing architecture descriptions is a current research topic, in which the attempt to recover the architecture from other artifacts, primarily from the source code of the system, is made. Invalid architecture descriptions contain errors which make a corresponding implementation impossible, cumbersome, hard to adjust or otherwise suboptimal. Even though the implementation may not have inherited these errors they were not corrected in the architecture description. Outdated architecture descriptions may be complete and sound, still they are of little use as they do not describe the corresponding implementation in its current state. In general there is a discrepancy between invalid or outdated architecture descriptions and their corresponding implementations. Different scientific approaches were undertaken to either avoid the problem completely (by tightly coupling architecture and implementation) or to detect mismatches between the software architecture and implementation. The latter approach can be further differentiated regarding whether the source code’s structure or the runtime behavior is compared to the described architecture. These approaches are referred to as static vs. dynamic analysis of a system.

This master thesis attempts to validate that an implementation matches its architecture description using a dynamic analysis of the evaluated system – the subject system. The taken approach is based on the principle of communication integrity which characterizes whether the communication inside a software system is performed only in ways intended by its architecture. The term communication integrity was defined by Luckham et al. as a property of a software system in which “the system’s components interact only as specified by the architecture” [LVM95b]. As part of the thesis an application named CommunicationIntegrityChecker (CIC) was developed. CIC uses a description of the architecture including the communication it allows and monitored execution data of the subject system as inputs. It determines whether a monitored execution represents a valid communication—that is communication that preserves the communication integrity property—and outputs the result. CIC consists of a class library implementing the needed functionality and two applications based on the library,
MonitoringTool focused on monitoring the execution data and InformationProcessor focused on evaluating the monitored data. While InformationProcessor requires the monitored data from MonitoringTool they are only loosely coupled as the data is published through the XMPP protocol in a chat room from where the InformationProcessor application reads the data. InformationProcessor itself publishes its results (whether a received execution was valid) for interested participants in the chat room as well.

1.2. Structure of This Thesis

The thesis is structured as follows: The introduction is completed by an introductory example which shows the problem statement as well as the graphical notation used throughout this thesis. At the beginning of chapter 2 important terms are defined and their relationship are described. Later on the importance of software architecture is depicted and it is analyzed how a software architecture description may lose its value over time. Chapter 3 summarizes the current state of research of topics related to the problem statement. Chapter 4 presents the concept developed throughout the work of this thesis. It includes the definition of entity types and their relations useful to describe real architectures and analyze them according to the problem statement. Chapter 5 summarizes the software CommunicationIntegrityChecker (CIC) designed and implemented to solve the problem statement according to the presented concept. Chapter 6 illustrates the value of both the conception and the resulting program by validating the execution of a real life software system. The thesis concludes with a summary of the solution presented.

1.3. Introductory Example

The following example is intended to communicate the problem statement in an easily comprehensible way. It also introduces the notation used throughout the thesis to describe architectures and their communication constraints. Consider the following simplistic architecture and implementation of a software system. The architecture consists of only three stacked layers (A, B and C) as shown in Figure 1.1a. The implementation is realized through the six classes (A1, A2, A3, B1, C1 and C2) depicted in Figure 1.1b whose details are omitted for the sake of simplicity. An architecture description of a software system always provides guidance and constraints on how to implement the software system. A layered architecture similar to Figure 1.1 in general carries the constraint that the entities of the layers may interact only with entities of their own or of adjacent layers. Even more the direction of the interaction is often predetermined as either top-down communication or bottom-up communication, meaning that an entity may interact only with other entities of layers below respectively above its own layer. Buschmann et al. distinguish top-down and bottom-up communication as different scenarios of layered architectures [BMR+96]. They state that top-down communication (Scenario I) is “probably the best-known
One would assume that the implementing classes follow the guidance and fulfill the constraints of the architecture. In this example each class must belong to exactly one layer and must obey the constraints of that layer. In this sense the classes are considered to be parts of the layers as shown in Figure 1.2a. This situation can be drawn using the Unified Modeling Language (UML) as shown in Figure 1.2b, with the layers depicted as packages. Still whether the layers are actually realized as packages or not depends on the specific implementation. From the implementational point of view the classes
can be seen as illustrated in Figure 1.3a. The six classes are ordered in a way similar to the architecture with thin, dashed lines which shall remind of the boundaries of the layers and their constraints regarding the interaction. This depicts a static view on the software system. A dynamic view on the software system does not depict the structure of a system but it does reveal the interaction that takes place in the system during runtime. Figure 1.3b gives an example of a possible sequence of interactions between (some) of the classes of implementing the layered architecture. The sequence consists of three messages send between instances of class A1, B1 and C1. Message (1.) is send from an object of type A1 to an object of type B1, message (2.) is send from the latter object to an object of type C1, finally the first object of type A1 sends a message (3.) to the C1 object\(^1\). All messages are answered with an return message which can be safely ignored for now. Having both the architecture description and actual interactions inside the software system allows to analyze the conformity between both. For example consecutively comparing the interactions of the sequence depicted in Figure 1.3b to the layered architecture given in Figure 1.2a yields to different results. The interactions executed by message (1.) and message (2.) correspond to the description of the architecture as they exhibit top-down communication between the adjacent layers A and B. The interaction executed by message (3.) however does not correspond to the architecture as it exhibits cross-layer communication.

Problems arise, as programming languages generally do not directly support architecture level entities nor enforce constraints between them. Specifically two different problems can easily be identified:

1. How to resolve which classes (or more generally: which code-level entities) belong

---

\(^1\)The name of the instances is of no interest for this thesis and neither are the actual sent messages which are therefore omitted.
1. Introduction

to which layer (or more generally which architectural entities)?

2. How to verify that the classes (code-level entities) obey the constraints of the layer (architectural entity) they belong to?

This thesis does not try to provide a general answer the first question. Instead it provides a modeling facility allowing to model both the architecture and the structure of the source code but requires a manually made mapping between code-level and architectural entities as an input. However it does provide an analysis facility to check whether the interaction of the code-level entities comply to the constraints of the architectural entities. As this approach analyzes the dynamic interaction inside a system at runtime—and not the static structure known before runtime—it forms a dynamic analysis of a software system.

Informal Notation

Throughout this thesis a simplistic and informal graphical notation featuring generic boxes is used instead of formal notations like the well-known UML. The informal notation, besides being more simple, emphasizes that the type of the architectural entity is of no importance to the solution of the problem at hand. The notation however supports explicit definitions of which entities are allowed to communicate with other entities. Figure 1.4 shows the introductory example in the informal notation. The notation features architectural entities for example layers as yellow boxes and code-level entities as blue boxes. It allows architectural entities to included code-level entities as well as other architectural entities by simply nesting them. Architectural entities including other architectural entities are not part of the introductory example but will be introduced in section 4.3. The notation also offers the ability to allow communication

Neither the layer type used in the example nor the package type used in the UML diagram is needed to define the entities and relations between them.
between architectural entities using green arrows and to explicitly deny communication using red arrows.
2. Background

Software architecture is considered crucial in the development of any software project as it plays several important roles, the first and maybe most obvious being that it serves as a basis for communication between persons responsible for designing and those responsible for implementing a system. A software architecture “establishes the design framework for the software designers and developers whose job it is to implement the system” [RM06]. More generally speaking, “software architecture can provide a vehicle for communication between the various stakeholders” [RM06]. It should be noted that nowadays a distinction between the architecture itself and descriptions of it referred to as architecture descriptions is commonly made. The ISO/IEC/IEEE 24765 standard defines software architecture as: “1. 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. 2. the organizational structure of a system or component. 3. the organizational structure of a system and its implementation guidelines. Syn: architectural structure” [ISO10]. All three definitions emphasize the high-level nature of software architectures (“fundamental organization, “organizational structure”) but only the first one describes what architectures actually consist of: “components” and “their relationships to each other”. This view of software architectures as components and their relations seems to be the common denominator for most of the definitions [LVM95a, RM06, GS93]. However these two parts are often further differentiated: the components resolve into the component itself and its public interface, the interface resolves further into required and provided interfaces, the relations resolve into connectors which may have their own semantics and connect components with matching interfaces. In addition other aspects like constraints and styles are supplemented to software architectures [Gar03]. The architecture descriptions may be informal like box-and-line drawings or formal when based upon an architecture description language (ADL, cf. section 3.1). Architecture descriptions, particularly formal descriptions, can be used for advanced analysis like throughput or deadlock detection aside from their mentioned role as communication means [GS93].
2. Background

2.1. Terminology

Some basic terms related to software architecture which are of particular importance for the scope of this thesis are described and defined as follows.

**Conceptual Architecture**

Software architecture describes a conception of a software system. In order to emphasize the conceptual nature of software architectures these sometimes are called *conceptual architectures*. When the term (software) architecture is used it usually refers to the conceptual architecture of a software. For example when discussing or establishing a new architecture in the design phase participants will always refer to the conceptual software architecture. Ducasse and Pollet define conceptual architecture as “architecture that exists in human minds or in the software documentation” [DP09]. Unfortunately the term conceptual architecture is not the only term used to name such architectures, the cited authors found “idealized, intended, as-designed or logical” being used synonymously. Furthermore architecture descriptions—when not generated from the source code—commonly describe conceptual architectures.

**Concrete Architecture**

Concrete architecture is the complementary term to the term conceptual architecture as explained above. It emphasizes that the root of a specific architecture is not a design activity but the resulting implementation itself. It also distinguishes the architecture as-implemented from the architecture as-intended (cf. conceptual architecture above), which should be equivalent but are often not. Ducasse and Pollet also define concrete architecture as “architecture that is derived from source code” [DP09]. Like conceptual architecture the term concrete architecture has its synonyms “as-implemented, as-built, realized, or physical” found by the cited authors.

**Architectural Gap**

After introducing the notion of conceptual and concrete architecture of a software system the question of their relationship arises. Ideally the conceptual and the concrete architecture of the same software system should be equivalent, that is the implementation of the system exactly resembles the conceptual architecture. However if the architectures are not equivalent a gap opens between them: the *architectural gap*. The term architectural gap again has several synonyms like “gap between the implementation and the intended architecture” [JSBT12], “gap between design and implementation” [MNMS01] or lack of “architecture compliance” [HRH+09].

**Communication Integrity**

One important aspect when reasoning about conceptual and concrete architectures of a software system and their (possible) architectural gap is the communication integrity
of the software system. Architectural gaps often manifest through the fact that components of the concrete architecture interact differently from the ways prescribed by the conceptual architecture. These are violations of the communication integrity of the software system as defined by Luckham et al.: “the system’s components interact only as specified by the architecture” [LVM95b]. This is intuitive as the architecture is the basis for the implementation of the system’s components as well as the guideline how they should interact. It is important to note that the term communication is used as the general term of any interaction, locally or distributed, by method invocation or through other mechanisms. Later Luckham et al. detail communication integrity with regard to architecture description languages as “components may communicate directly only if there is an architecture connection between their interfaces” [LVM95a]. The concept of communication integrity is central for this thesis. It is concerned with the question of how violations can be detected and what prerequisites to analyze the communication integrity are.

Hierarchy of Architectures

Conceptual architectures of the same software system may differ in their grad of abstraction. This is sometimes called the hierarchy of architectures which goes down from abstract to more concrete architectures. An architecture hierarchy “is a linear sequence of two or more individual architectures that may differ with respect to the number and kind of components and connections among them” [MQR95]. Such hierarchies are useful to provide more comprehensible views through the more abstract architectures and more detailed views through the less abstract architectures: “In general, an abstract architecture is smaller and easier to understand; a concrete architecture reflects more implementation concerns” [MQR95]. As the cited meaning of the term concrete architecture conflicts with the previously introduced it is avoided for the rest of this thesis and the more general term hierarchy of architectures is used instead.

Subject Software System

The main aspect of this thesis is to analyze software systems with regard to their communication integrity property. This requires reasoning of their conceptual and concrete architectures as well as of their architecture descriptions. The term subject software system represents as software system that shall be validated yet not be specified in detail as its specifications are of no particular interest of this thesis. The term subject software system is therefore used in the following sense: The software system to be analyzed and which the architectures and architecture descriptions belong to.

Software Evolution

Software systems like everything else age. However, unlike other objects, in particular living organisms, software does not change on its own and hence would always be as
helpful as on its first day. Generally this is not the case, rather software evolves over its lifetime as it is changed to adopt new requirements \cite{Par94}. This so-called **software evolution** is “the sequence of changes to a software system over its lifetime; it encompasses both development and maintenance” \cite{CJH00}. The reason for this evolution is the continuing change of requirements for almost\footnote{Lehman distinguishes between three types of programs and his first law only applies to programs of the “E-type”. However almost all real-world software systems are of this type.} all programs. This is known as Lehman’s first law of software evolution \cite{Leh96}:

“Continuing Change. An E-type program that is used must be continually adapted else it becomes progressively less satisfactory \cite{Leh96}.”

Lehman’s second law of software evolution \cite{Leh96} states the consequence of this continuing change:

“Increasing Complexity: As a program is evolved its complexity increases unless work is done to maintain or reduce it \cite{Leh96}.”

However Lehman does not take into account the architecture of the software. The continuing change and the increasing complexity may cause architecture descriptions and their implementations to drive apart, or, as Murphy et al. describe, to drift apart:

“Most artifacts, however, are manipulated independently of one another, even if they are logically related; design documents and source code are an example. When artifacts are independently manipulated, the artifacts tend to 'drift' apart over time.” \cite{MNMS01}
3. Related Work

If I have seen further it is by standing on the shoulders of giants.

---

Isaac Newton

The importance of the architecture and architecture descriptions for successful realization has already been described in section 1.1. The common problems regarding architecture descriptions which may be missing, incomplete, (partly) invalid or outdated have been stated there as well. All these problems lead to a discrepancy between the conceptual or intended architecture and the concrete or realized architecture or simply spoken the implementation. This architectural gap mitigates the software quality and may lead an “architectural erosion [that] can transform software architectures into unmanageable monoliths” [TVBC12]. Architectural erosion can transform software architectures into unmanageable monoliths. In the following several techniques using different approaches to close the architectural gap are presented.

3.1. Architecture Description Languages

There is an obvious need to describe software architectures in order to allow documenting and communicating them. The earliest, still common, attempts to describe architectures used graphical, yet informal depictions\(^1\). These informal drawings are often called box-and-line drawings and "have for a long time been the only means for describing SAs" [MLM\(^+\)13]. However as box-and-line drawing have many shortcomings originating mostly from their ambiguity many, more formalized, attempts were made to advance the ability to describe software architectures. Malavolta et al. use the term architecture language (short AL) as a generic term to cover all kinds of approaches to described architectures and further differentiate: “Classically, ALs have been classified into three broad categories: box-and-line informal drawings, formal architecture description language, and UML-based languages” [MLM\(^+\)13].

---

\(^1\)The informal notation used throughout this thesis is an example for such informal depictions.
3. Related Work

Languages (short ADLs) are usually considered to be “formal languages that can be used to represent the architecture of a software-intensive system” [Cle96]. The formalization of ADLs should allow unambiguous definition of software architectures and by this increase the usefulness of architecture descriptions in documenting and communicating. Furthermore it should allow deeper analyses of architectures. However it is still being discussed what features ADLs must provide. Medvidovic and Taylor point out that the features may range from “simple, understandable, and possibly graphical syntax, well-understood, but not necessarily formally defined semantics, and the kinds of tools that aid visualization, understanding, and simple analyses of architectural descriptions” to “provide formal syntax and semantics of ADLs, powerful analysis tools, model checkers, parsers, compilers, code synthesis tools, runtime support tools, and so on” [MT00]. This roughly matches the distinction of first and second generation ADLs with the former originating in the 1990s and the latter emerging after the year 2000 [MLM+13]. Examples for first generation ADLs include ACME, C2, Darwin, Rapide, UniCon, Wright and many others which were surveyed by Medvidovic and Taylor in 2000 [MT00]. All of the former and more recent ADLs belonging to the second generation ADLs like SOFA, PiLar, PRISMA, COSA, AADL and CONNECT were analyzed by Ozkaya and Kloukinas. The sheer number of proposed ADLs indicates the enormous importance ADLs have gained in scientific research over the last twenty years. Yet despite the many suggested solutions ADLs have not made their way into the industry. It has been reported that ADLs were not adopted by the industry because of “the mismatch between the way architects work and the support provided” by ADLs [MLM+13]. Even more the great amount of suggested ADLs and the emerging insight that there will not be a one-size-fits-all ADL is not likely to support the adoption of ADLs in the industry. On the other hand UML is “a de facto standard general modeling language for software developments” [RKJ04] which might lead software architects to use UML for architecture descriptions as well. Ozkaya and Kloukinas in any case state that “practitioners insist on using UML even though it is known that UML has very weak support for architecture specification (e.g., no first-class connectors, no formal semantics, etc.)” [OK13].

In the scope of this thesis the question arises whether ADLs provide an alternative to the presented approach. More specifically can ADLs close the gap between the intended/conceptual architecture manifested in its description and the realized implementation? Some ADLs provide the ability to generate source code from it architecture models. Such source code equates, at least theoretically, the architecture descriptions it was generated from. However ADLs are not targeted to supersede programming languages and therefore even generated source code in general is modified after its generation. This allows gaps between realization and architecture description to emerge after the initial creation of the source code. In general ADLs do not provide a tight coupling between modeled architecture and realizations which prevent them from detecting the architectural gap.

Still ADLs might provide at least the model of an conceptual architecture which is

\(^2\)“In summary, it is clear that an ideal and general purpose AL is not likely to exist” [MLM+13]
part of the required input for the approach taken in this thesis. It seems desirable to use an accepted modeling approach together with its tool support instead of propagating a new one which requires new tools. However the marginal adoption of ADLs in the industry, which is even more scattered across the many available ADLs, would lower the helpfulness. The dominance of UML for modeling software in the industry together with its tools support makes it intriguing to integrate UML in the chosen approach. Whether this can be accomplished—given the fact that UML is often considered to be insufficient to model architectures as mentioned before—is not yet clear but it might be a worthwhile exploration in the future.

3.2. Connecting Architecture and Implementation

The prerequisite of both dynamic and static analysis of the conformity between architecture descriptions and their realization is to link them. While, as mentioned before, architecture description languages generally do not provide such functionality this shortcoming has already been detected [BW09, ACN02a]. Two fundamentally different approaches to overcome this obstacle are presented as follows: The first is a new ADL named LISA which was designed to enable “description of high-level architectural elements like components and systems and of low-level concepts like classes and interfaces” [BW09] and connecting both. ArchJava presents a second, quite opposite, approach in not linking code-level entities like classes to architectural entities but instead “unifying architectural structure and implementation in one language” [ACN02a] as to allow defining architectural entities directly inside the source code.

LISA

LISA (Language for Integrated Software Architecture) consists of an LISA-ADL which is an “extensible XML-based language for representing structural relationships of heterogeneous software systems” and the LISA-toolkit which provides the required tool support. Figure 3.1 gives an overview of LISA. The LISA-ADL provides a metamodel called the “LISA Architecture Model” and technology bindings which enable linking the abstractions form the architecture to their (technology-dependent) realization. The “LISA Architecture Model” consists of the layers “Basic Structure”, “Component” and “System” with increasing abstractions. Some of the low-level abstractions as classes or packages from the “Basic Structure Layer” can be retrieved directly from the source code while other abstractions like layers or modules must be defined manually. The next higher layer provides support to components. Components implemented (directly) using object-oriented languages can be defined with entities of the “Basic Structure Layer” while components using a component model are to be defined with entities of the “Component Layer” and technology bindings which support “EJB, Spring, OSGi, Spring Dynamic Modules, and SCA” [BW09]. Finally the “System Layer” allows to model whole systems and further allows hierarchical decomposition of these.

The LISA-toolkit is the provided tool support of the LISA. It consists of several
3. Related Work

plug-ins for the Eclipse platform [Ecl14a]. They include support to model, visualize and validate architectures as well as to synchronize architecture descriptions with their implementations. Synchronization requires mapping between the architecture description and the implementation. Figure 3.2 gives an example of how such an mapping is visualized using the LISA-toolkit. The given example is an SCA-based project consisting of two classes, one interface and an SCA configuration file. These are mapped onto a system consisting of only one module “ordering” which again consists of only one layer “core”. The layer consists of the two components “Account” and “OrderProcessing” together with the contract “AccountContract”. These are mapped by the classes and the interface of the source code. Finally the SCA configuration file
is mapped onto the system definition “System”. All the technology bindings are SCA bindings as the elements of the source code are realized as SCA elements. While all architectural entities are mapped in this example—which is observable by the classes (the interface) below the components (the contract)—it is not required by LISA to map all entities immediately. This allows for designing the architecture of a new system (or extending an existing architecture) without implementing it beforehand. In general LISA supports both top-down and bottom-up approaches to define an architecture for a new or an already existing software system while also “a combination of both approaches can be used” [BW09].

The type of errors or violations the LISA toolkit can detect depends on the layer. On the lowest layer, the “Basic Structure Layer”, differences between implementation and architecture entities and missing mappings are detected while on “the level of component and system modeling [...] missing connections, unset property values, and layer violations” are detected [BW09]. The detected problems are visualized in different views of the LISA-toolkit. The same architectural violation is depicted in Figure 3.3 inside two different views. Figure 3.3a shows the violation as a red connection (in opposition to the normal black connections) between two components while the same violation yields a marker together with an overlay describing the violation in the source editor.

It should be noted that LISA does not provide an alternative textual notation to its XML definitions as its authors expect that “LISA-based architecture models are created and manipulated visually using the LISA-toolkit” [BW09]. Furthermore the authors of LISA also thought about how to integrate LISA into the software life cycle [WB12]. They state that LISA offers “comprehensive support for architecture-related activities during the whole software life cycle” [WB12]. While the LISA approach appears promising it seems to not have gained much attention from other researchers. Also the license and

---

Figure 3.3.: Two views of the same violation [BW10]

---

3The central paper about LISA is—according to the ACM Digital Library and Google Scholar—cited only two respectively three times by authors other than the original ones.
3. Related Work

the source code of the LISA-toolkit are not revealed.

ArchJava

While the former described concept of ADLs enables formal definition of architectures the resulting architecture descriptions only provide guidance to the developers about how to implement a software system. More specifically architecture descriptions (based on ADLs or not) of an software system are generally not linked directly to the source code of the same system. This allows the architectural gap to emerge as said before. ArchJava [ACN02a] takes a different approach by defining the architecture directly inside the same artifact as the realization which is the source code of a software system. In order to allow this ArchJava extends “a practical implementation language to incorporate architectural features and enforce communication integrity” [ACN02a]. As ArchJava name indicates the extended implementation language is Java. ArchJava adopts the concepts of components, connections and ports from ADLs but allows (or requires) to define them as part of the source code. The example depicted in Figure 3.4 taken from the initial article presenting ArchJava, describes a simplistic compiler. The compiler’s architecture consists of the three components scanner, parser and codegen which are sequentially connected through ports.

The parser component’s exemplary implementation given in Figure 3.5a shows how a component (line 1) and its in and out ports (line 3,8) are defined using the newly introduced keywords component and port. A port functions as “a logical communication channel between a component and one or more components that it is connected to”. It consists of required and provided methods defined by the new keywords requires and provides. Provided methods of a port must be implemented by the port’s component and may be called by other components connected to the port’s component. These connected components conversely require these methods through their port. The parse method of the parser component gives an example by calling the nextToken method required by its in port (line 13). Consequently a component—more specifically the scanner component as seen in the architecture description given in Figure 3.4—must provide a nextToken method with the same signature through its out port.

The compiler component listed in Figure 3.5b serves as an example of a component consisting of subcomponents as well as of how to connect components through their ports in ArchJava. The Compiler component consists of the three subcomponents (line 2,3,4) given in Figure 3.4 and also interconnects them (line 6,7) accordingly using the keyword connect newly introduced by ArchJava. The source code listing also reveals that components may directly communicate—that is not through a connection—with their

\(4\)The LISA approach is the most notable exception to this.
3.3. Static Analysis

Fraunhofer SAVE (Software Architecture Visualization and Evaluation) “is a tool for analyzing and optimizing the architecture of implemented software systems” [DKL09].
3. Related Work

The SAVE approach initially supported static analysis by comparing architecture definitions and source code of software systems [KLMN06]. This “Static architecture compliance checking” [DKL09] is based on the idea of reflexion models introduced by Murphy et al. [MNMS01] and extended to hierarchical reflexion models by Koschke and Simon [KS03]. Figure 3.6 gives an overview about reflexion models and depicts each of the three comparison results.

Reflexion models consist of two views: the hypothesized view which represents the architectural view and the concrete view which represents the source code view on a software system. Conceptual entities reside on the hypothesized view and represent “conceptual modules” while concrete entities represent modules or entities of the source code. Mappings between the conceptual and concrete entities span across the border of the two views and connect modules of the architecture and the implementing source code. Finally reflexion models feature references between two entities of the same type, that is exclusively between two conceptual entities or between concrete entities, which mark “abstract dependency relations”. Examples for such abstract dependencies in the concrete view are “calls, variable accesses, and type dependencies”. In order to perform the architecture compliance checking entities and references on the hypothesized and the concrete view are compared. Each comparison yields to either a convergence, divergence or absence which are defined as follows:

Convergences are references in the hypothesized view also present in the concrete view.

Divergences are references in the concrete view for which no reference in the hypothesized view exists.

Absences are references in the hypothesized view not present in the concrete view.

---

5 This reflects the terminology used by Koschke and Simon which they aligned to “IEEE recommended practice for architectural description of software-intensive systems” but also provides a mapping between their and Murphy’s original terminology.
The appliance of SAVE requires several steps of which some are automated while others must be performed manually [LM08]. Still SAVE provides graphical tools for some of the manual tasks. First the “planned”, that is the conceptual, architecture is modeled by experts (e.g. software architects or lead developers of the software system to analyze) using an simple, UML based editor. The second step is to extract the actual, that is the concrete, architecture. This step is automated and does not require manual effort. The next step to enable a mapping between the entities of both architectures which must be done manually but is supported by a simple mapping editor. This allows to automatically compare both architectures. This comparison reveals the convergences, divergences and absences which are depicted as checkmarks, exclamation marks and Xs while question marks represent different, aggregated results. Figure 3.7 exemplifies an output containing all possible results. The last two steps are again manually executed and consist of assessing the criticality of each divergence or absence by an expert and removing the critical deviations.

3.4. Dynamic Analysis

The former section described an approach of static analysis of software systems. It seems that more research effort is being committed to static analysis than to dynamic analysis. This might reflect the point that the static view of a software system is considered to be of higher value than the dynamic view. This is questionable, to say the least, as Dragomir et al. state: “the dynamic view of the architecture is often neglected, although it is crucial to understand and further evolve the considered system” [DL12]. Particularly for loosely coupled systems e.g. systems with an service-oriented architecture (SOA) the dynamic view is of outstanding importance as “the interplay of the various service-providing elements is more important than their mere static structure” [DL12]. DiscoTect [YGS+04] and Kieker [vHRH+09] as two of the existing approaches of dynamic analysis are briefly presented as follows.

**DiscoTect**

DiscoTect presents a “technique is to determine the architecture of a system by examining its behavior at run time” [YGS+04]. DiscoTect allows to reconstruct an architecture based on a software system’s behavior. The main problem in this approach is to “bridge the abstraction gap” [YGS+04] that is to connect observations made during runtime with architectural decisions. DiscoTect attempts to accomplish this bridging with the definition of styles both for the realization (implementation styles) and
3. Related Work

the architecture (architecture styles) which are to be mapped. The implementation styles—which are conventions of the source code—are used to identify significant information extracted from the execution trace of software system. The architectural styles on the other hand are used to identify architectural structures and operations. The mapping between both is done by state machines as several operations of the implementation styles normally assemble one architectural style. The overall course of actions done by DiscoTect is given in Figure 3.8: A trace engine provides runtime information from a software system. This information is then converted to implementation styles which might cause a state change in a state machine implementing the mapping between implementation and architectural styles. Occasionally such a state change yields a new architectural style which is passed to an architecture builder which adds it to the model of the software system. The authors of DiscoTect claim to be able to successfully discover architectures using their approach and document this with a case study on a specific subject software system. They used the discovered architecture to manually compare it with the architecture descriptions of the subject software system. The comparison revealed four different types of discrepancies between the conceptual and concrete architecture. However DiscoTect does not feature modeling of the conceptual architecture and by this is unable to automatically compare architecture description with their discovered architectures.

Kieker

Kieker is a monitoring framework that allows “continuous monitoring of software services” [vHRH+09]. The term continuous monitoring emphasizes that the monitoring is executed not only during the construction of the software for debugging or profiling means but instead continuously and in particular during the normal operation of the software system. This requires especial light-weight monitoring facilities which impose only a very small performance overhead. The authors of Kieker systematically measured the overhead of their framework and conclude that it “imposes only a small linear overhead, and that it is applicable in industrial settings”. The tracing of single method executions is done by so-called monitoring probes. Several monitoring probes based on different technologies are part of the framework. First of all Kieker provides monitoring probes based on AspectJ [Ecl14b]. These allow to weave the monitoring probes into to subject software system on compile time using annotations or at load time without any manipulation of the source code. This allows for unobtrusive monitoring of all
non-distributed Java-based software systems. In order to monitor distributed software systems, Kieker provides monitoring probes “employing AspectJ [10], Java EE Servlet [7], Spring [8], and Apache CXF [9] technology” [Kie13]. However, currently, Kieker is bound to Java technologies and cannot monitor implementations in other programming languages. Kieker provides two representations of the trace information gathered by its monitoring. The first representation called execution trace “is simply the ordered (by execution order index values) sequence of executions” [vHRH+09]. The second representation called message traces provide more detailed information of the (potentially nested) invocation of methods and their returns. A message trace consists of an ordered sequence of messages which might be call or return messages. Messages of both types normally have a sender and a receiver but one of them might be missing if the sender or receiver was excluded from the monitoring. This supplies not a simple chronological sequence of the invocations but hierarchical call trees based on the method invocations. Figure 3.9a shows an UML sequence diagram with an example message trace consisting of four call messages and their return messages. The ordering of the messages hereby reveals the call’s hierarchy as observable from the successive call messages getOffers and getBooks. Figure 3.9b gives an overview of how message traces are realized within Kieker.

Kieker provides “several visualizations of a system’s runtime behavior, such as UML sequence diagrams, dependency graphs, and Markov chains” in order to analyze the monitored traces. These analysis can be performed either online (during execution) or offline (after termination). The authors of have stated they intend to “compare prescriptive architectural models, which are designed in forward engineering, with architectural models reconstructed based on monitored runtime

---

6In order to use AspectJ annotations (like for all annotations) version 5 or higher of Java is required
3. Related Work

information”. However currently Kieker does not provide a metamodel of software architectures and can therefore not automatically compare its result to an described architecture. Furthermore the current analysis emphasizes the visualization of the dynamics of a software system and not the retrieval of an architectural model. However due to its powerful execution monitoring and its open-source licensing Kieker is an excellent framework to extend with further analysis abilities or an excellent monitoring basis to build analysis tools upon as within this thesis.
4. Concept

Perfection is achieved, not when there is nothing more to add, but when there is nothing left to cut away.

ANTOINE DE SAINT-EXUPÉRY

This chapter presents the general concept developed during this thesis. It relies on the information that is summarized in chapter 2. It consists of a simple metamodel of software architectures that are to be analyzed and procedures to validate them during run time. In order to allow validation of all kinds of software systems this thesis makes only very few assumptions about the architectures of these software systems. These assumptions are very similar to the following characterization by Reekie and McAdam which is one important definition of software architecture:

1. The whole consists of smaller parts.
2. The parts have relations to each other.
3. When put together, the parts form a whole that has some designed purpose and fills a specific need.

The “smaller parts” in (1.) are named architectural entities and will be described in section 4.2. The relations among them (2.) are identified as include relations and communication rules, described in section 4.3 and section 4.4 respectively. Finally while there surely exists a “purpose” of the “whole” (3.), it can be safely ignored for the validation objective (no matter whether the “filled needs” are architectural or application-related). Instead, the needed functionality must be connected to the
architecture in order to achieve the intended functionality. This is done by special “parts”
named code-level entities introduced in section 4.1.

By assuming only these few conditions the creation of a very generic metamodel of
architectures is enabled. This generic metamodel shall allow the creation of models
for most (if not all) architectures of systems to be validated. The small amount of
assumptions also keeps the metamodel straight and simple.

4.1. Code-Level Entities

In the introductory example from section 1.3, six classes represent the implementation
of the target system. Such representatives are needed in order to model target software
systems as a whole consisting of both: entities of the architectural level of the software
system and entities of the source code implementing the software system. Apart from
classes, programming languages provide other types of entities which may represent
the implementation. Table 4.1 shows code-level entities of three of the most used
programming languages C, C++ and Java. The type of the programming language

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>C++</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions/Methods</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>Structures</td>
<td>y</td>
<td>y</td>
<td>n</td>
</tr>
<tr>
<td>Classes</td>
<td>n</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>Packages</td>
<td>n</td>
<td>n</td>
<td>y</td>
</tr>
<tr>
<td>Namespaces</td>
<td>n</td>
<td>y</td>
<td>y</td>
</tr>
</tbody>
</table>

Table 4.1.: Code-level entities

which can be logical, functional, procedural or object-oriented is the dominant factor
that determines which entity types a programming language provides, e.g. only
object-oriented languages like C++ or Java support classes. Still languages of the same
type can differ, as for example C++ does not have a package concept while Java does.
Many entity types allow to embed other entities, e.g., namespaces are designed to embed
all of the other mentioned code-level entities even other namespaces. Each individual
entity (with the exception of anonymous entities which can only be accessed at the place
of their definition) has a name through which it can be accessed in the context of its
definition. Moreover the entities must be identifiable through a unique identifier if they
are to be used in contexts different from their definition. That unique identifier is often
called the full qualified name of the entity.

Constraining the representatives to a particular type like classes is not desirable as it
requires potential target systems to be implemented using this particular type\(^1\). Instead,
allowing entities of arbitrary types grants a high flexibility in defining representatives.

\(^1\)For example allowing only classes to represent the implementation implies constraining target software
systems to systems implemented through object-oriented programming languages.
This thesis uses **code-level entities** as representatives instead of predefining the type of the entities representing the implementation. Code-level entities only consist of a unique identifier and a mapping to the actual entities which may be of arbitrary type\(^2\). While the actual type (e.g. class, method or function) of an entity can often be neglected, an entity is required to have a unique identifier (a full qualified name) to allow it to be mapped on a code-level entity entity. Code-level entities are used as building blocks to model the entities of the actual source code as simple, generic yet identifiable building blocks without additional information like type or semantics of the entity. In the informal notation used throughout this thesis they are depicted as blue boxes as shown in Figure 4.1. In order to increase readability these blue boxes may be depicted similarly to UML class entities, if they represent exactly one class like in the introductory example. Code-level entities can also be omitted in illustrations of higher abstractions where they do not contribute to the comprehensibility.

Code-level entities represent the entities of the implementation and serve as aggregations of these entities—independent of their types—which can be identified by a unique name. As such, code-level entities form mappings between a unique name and a set of entities of the source code implementing the target software system. An exemplary mapping of the code-level entity “Entity1” to three entities of the implementation is shown in Figure 4.2. The mappings formed by the code-level entities need to be unique—that is no building block is mapped more than once and each name

\(^2\)Code-level entities can be seen as simplifications of class entities in UML diagrams as they do not have a type nor members but still have a name. Classes themselves are excellent candidates to be mapped by code-level entities.
of a code-level entity is unique—to allow unambiguous identification. The particular
definitions of code-level entities may be fine grained, that is a 1-to-1 mapping to functions
or methods, or coarser grained, that is mappings of one or more classes (or other entities
like packages or namespaces) with their respective functions or methods. The latter is
quite similar to the import functionality in the Java programming language (as shown
in listing 4.1) in which a (sub) package with an asterisk identifies all code-level entities
embedded by the package.

```java
// import all classes, interfaces, enums embedded by subpackage
// (which is embedded by superpackage)
import superpackage.subpackage.*;
```

Source Code 4.1: Import of all entities embedded by a (sub) package

In summary, code-level entities can be used as atomic building blocks to model the
implementation of a target software system. Three assumptions about code-level entities
are made in the scope of thesis:

- Code-level entities have a unique name and can be identified through this name.
- Code-level entities are atomic which means that they do not include other
code-level entities (or architectural entities introduced below).
- Code-level entities provide unique mappings to building blocks of the
implementation.

### 4.2. Architectural Entities

Architectural entities are the building blocks to model the software architecture of a
software system. The types of entities available for modeling depends on the modeling
language used. For example UML provides classes, components and packages while other
modeling languages may offer a different set of entity types. Even more, architectures
can be drawn without the use of a (formal) modeling language in an informal way like
in the architecture sketch of the introductory example at the beginning
of this thesis. In the scope of this thesis architectural entities do not have a type
unlike in formal modeling languages. While omitting types of code-level entities as
described in section 4.1 grants independence from programming languages, omitting
types of architectural entities grants independence from modeling languages and allows
modeling architectures based on informal architecture descriptions. In order to identify
architectural entities they need a unique name. Architectural entities are depicted as
simple yellow boxes as shown in Figure 4.3. Architectural entities in general can represent
aggregations of other architectural entities of a lower abstraction level. For example a
package entity in an UML diagram usually stands for several classes. Other architectural
entities like UML class entities represent exactly one entity. The architectural entities

3From the modeling point of view code-level entities are atomic even if they actually identify a set of
entities of the source code.
developed in this concept can always represent other architectural entities. In order to connect the architecture to the implementation, architectural entities can also represent code-level entities. The representation of other entities is depicted by nesting the boxes of the entities as shown in Figure 4.3b and Figure 4.3c. In the following the relation between entities nesting other entities is called include relation. Include relations as part of the elaborated concept are developed further in section 4.3. Apart from representing or including entities architectural entities might have relations to other entities like dependencies or associations in UML. These relations are the main source of communication rules that enable definition of allowed communication between architectural entities. The concept of communication rules is described in detail in section 4.4.

4.3. Include Relations (∈)

Architectural entities are not only building blocks for architecture descriptions. They also aggregate other entities which can be other architectural entities or code-level entities. Thus there is a kind of an include relation between architectural entities and code-level/architectural entities. This include relation is often depicted as one entity embedding another entity, while formal languages also define an explicit include relation. Naturally the include relation is directed as the opposite—interpreting A includes B as the same as B includes A—is paradoxical. Furthermore the include relation is acyclic as an entity cannot include itself (not even through other included entities). Lastly each architectural entity is included by at most one other architectural entity. In summary the include relations of architectural entities form a forest of include trees (or mono-hierarchies). Include tree always consists of architectural entities as root and inner nodes. Code-level entities are considered atomic on the modeling level (cf. section 4.1) and therefore cannot include other entities (and not be inner nodes of include trees). Instead code-level entities should form the leafs of each include tree in order to connect the architecture with the implementation. This however is not enforced to allow modeling architectures before their realization are finished. In order

4It is even antisymmetric as its impossible for A to include B and be included by B at the same time.
to illustrate the include trees induced by the include relation the introductory example is extended by adding new architectural entities of a higher abstraction level than the former layers. The six classes representing the implementation persist and are included in the three layers as before. The layers however are now included as a whole in a new entity called “Controller” and two new separate entities “Model” and “View” are added to create a Model-View-Controller (MVC) architecture. The example is depicted in Figure 4.4 showing also the typical associations of the MVC architecture pattern\(^5\). Analyzing the include relations of the architectural and code-level entities depicted in the

MVC example yields an include tree of the “Controller” entity as shown in Figure 4.4b. The “Controller” entity is the root node of the tree as it is not included by any other entity. The entities “A”, “B” and “C” are children of this root node, as they are included by the “Controller” entity. Each of the six code-level entities is a child node of one of the architectural entities “A”, “B” and “C” according to their include relations.

Include relations are of particular importance in the scope of thesis and therefore a special notation to define include relations is introduced in the following. The fact that one entity A includes another entity B may also be phrased as “B belongs to A” or “B is part of A”. Inspired by the theory of sets include relations are—in the scope of this thesis—symbolized as \( B \in A \) with the meaning that A (directly) includes \( B \)\(^6\). It should be noted that is the only mathematically adequate if the entities are interpreted as sets. That implies for the MVC example:

\(^5\)These associations are not relevant for the include relations and can be safely ignored.

\(^6\)The symbol \( \in \) was chosen to emphasis the intuition that an included entity is part of another entity.
Controller := \{A, B, C\}

\begin{align*}
A & := \{A_1, A_2, A_3\} \\
B & := \{B_1\} \\
C & := \{C_1, C_2\}
\end{align*}

In order to maintain accessibility a more relaxed approach is followed and \(\in\) is treated as a relation between entities as described before.

### 4.3.1. Indirect Include Relations \((\in^*)\)

Include relations as described before can be (and usually are) nested. These nested includes form indirect includes. That is if “B is part of A” and there is an entity C with “C is part of B” then “C is an indirect part of A”. This adds a kind of transitivity to include relations. However in this thesis it is required to measure the number of includes between two entities instead of only use the transitivity. In order to allow this, the include relation from above is extended in the following way:

Using the direct include relation:  
\[ x \in^1 y \iff x \in y \quad (4.1) \]

And an inductive rule:  
\[ x \in^{i+1} y \iff \exists x' x \in x' \land x' \in^i y \quad (4.2) \]

By this we gain a measure for the level of inclusion which we call in analogy to graph theory the distance between the including and the included entity. The distance \(i\) can be replaced by * which stands for an arbitrary distance when the concrete value is not of interest\(^7\):

\[ x \in^* y \iff \exists i \in \mathbb{N}_{>0} : x \in^i y \]

\(i\) can also be replaced by + to denote only indirect includes\(^8\):

\[ x \in^+ y \iff \exists i \in \mathbb{N}_{>1} : x \in^i y \]

### 4.4. Communication Rules \((\rightarrow)\)

Architectures, as said before, consist of parts and relations. The parts—identified as code-level and architectural entities—were already covered by section 4.1 and section 4.2. The first type of relation, the include relation, was described above in section 4.3. In this section the second and last relation type between entities is presented. This relation generalizes concrete relations like associations, inheritance or dependencies.

---

\(^7\) It should be noted that in the context of include relations * stands for numbers \(\geq 1\) not 0 as commonly used.

\(^8\) It should be noted that in the context of include relations + stands for numbers \(\geq 2\) not 1 as commonly used.
4. Concept

in architecture descriptions. For example an association between two classes allows instances of one class to use instances of the other class i.e. to call methods of the other class through its instances. Another example is the specialization relation which allows classes to directly utilize (public or protected) methods of its super classes. Eliminating the semantics (e.g. the is-a semantic of a specialization) of such relations leaves only the statement that one class may utilize the other class. Generalizing concrete relations like associations strips off the semantics of the relation as well as the involved types (e.g. classes) and instead allows defining which entities may interact. These relations are named communication rules or allowing communication rules as they explicitly state that one entity is allowed to interact or communicate with another entity. Communication rules are derived from the architecture description of a software system. Therefore the relationship is defined solely between architectural entities. It is also directed in that allowing one entity to communicate with another does not imply that the latter is allowed to communicate with the first. Analyzing the introductory example (cf. Figure 1.1a) from the first chapter of this thesis in order to retrieve communication rules yields:

- Layer A may communicate with layer B.
- Layer B may communicate with layer C.

Similarly to the include relation the relation formed by communication rules is directed. Communication rules cannot, in contrast to the include relation, be generally considered acyclic. This can be recognized easily when removing the constraint to allow only top-down communication in the introductory example: Allowing communication downwards and upwards the layers immediately yields cycles between the adjacent layers (A-B, B-C) and one cycle covering all layers (A-B-C-B-A). It can be convenient to utilize communication rules to explicitly deny interaction between entities (as detailed in the next section). In that case communication rules carry a permission attribute which has one of the two possible values: ALLOW, DENY. Rules carrying the DENY permission are called disallowing communication rules and are not discussed here but later on in section 4.5. Without an explicit permission communication rules always imply allowed communication. Allowing communication rules are symbolized as $A \rightarrow B$ with the meaning that $A$ may communicate with $B$ while $A$ and $B$ are architectural entities. In the informal notation used throughout this thesis (allowing) communication rules are depicted as thick, green arrows as shown in Figure 4.5. These arrows represent

\[ A' \quad \rightarrow \quad B' \]

Figure 4.5.: Communication rules:

\[ B' \rightarrow A' \]
\[ A' \rightarrow B \]

\[ A \]
\[ B \]

\[ \text{The architecture description can be communicated as formal models, informal architecture sketches or even verbally. Also combinations of these are possible.} \]
rules which allow communication from the source entity of the arrow to its target entity. It should be noted that communication rules may exist between entities which are included by other entities. Such cases are depicted by arrows starting and ending at the entities affected by the communication rule and overlapping any including entities. The depiction of rule $A' \rightarrow B$ shown in Figure 4.5 is an example for such an overlapping arrow.

**Derived Communication Rules ($\rightarrow^*$)**

After describing both, the include relation and communication rules, the question arises whether these relations affect each other. Figure 4.6a gives an example of two entities both including another entity and being part of the same communication rule. In Figure 4.6b the same situation is modeled emphasizing the includes of the entities entity.

In general communication rules defined on architecture entities which include other architecture entities intuitively imply that the rules apply to all included entities. So in a sense rules can be *derived* from other rules together with matching includes. These rules are therefore named **derived communication rules**. In the informal notation used throughout this thesis derived communication rules are depicted similarly to the original communication rules but in a paler green and with dashed outline to differentiate them.

Augmenting the initial situation shown in Figure 4.6 with all derivable communication rules leads to a situation as depicted in Figure 4.7. In order to get a clear definition of derived communication rules they are elaborated more accurately in the following. A derived communication rule is a communication rule constructed by replacing one the entities affected by that rule with one of the entities it includes. In order to enable derived communication rules the original relation describing
communication rules is extended in the following way:

Using the not derived communication rule:

\[ x \xrightarrow{0.0} y \iff x \rightarrow y \]  

and an inductive rule for the left side:

\[ x \xrightarrow{i+1,j} y \iff \exists x': x \in x' \land x' \xrightarrow{i,j} y \]  

and an inductive rule for the right side:

\[ x \xrightarrow{i,j+1} y \iff \exists y': y \in y' \land x \xrightarrow{i,j} y' \]  

At most times it is of no particular interest whether the include relation is followed on the left or right side of the communication rule. Therefore the following shorthand is useful.

Shorthand:

\[ x \xrightarrow{k} y \iff \exists i \exists j: i + j = k \land x \xrightarrow{i,j} y \]  

The number \( k \) is than called the **derivation number** of the communication rule. Using the value of indirect include relations as introduced in section 4.3 derived communication rules can equivalently be defined as:

\[ x \xrightarrow{k} y \iff \exists i \exists j: i + j = k \land x \xrightarrow{i,j} y \]  

As sometimes the concrete value of the derivation number is negligible \( k \) can be replaced with the placeholders \( * \) and \( + \) standing for any \( k \in \mathbb{N}^0 \) or any \( k \in \mathbb{N}^+ \) respectively. \( x \xrightarrow{*} y \) identifies all derived communication rules between \( x \) and \( y \) while \( x \xrightarrow{+} y \) identifies all communications rules between \( x \) and \( y \) regardless of whether they are derived or not.

**Ambiguous Communication Rules**

Communication rules can be ambiguous in the following sense: Two rules are ambiguous if they apply both to the same pair of entities. Defining the same rule twice obviously results in two ambiguous rules. Defining rule \( A' \rightarrow B' \) is defined twice. Even more deriving communication rules as described above may result in additional ambiguous communication rules which were not defined beforehand. The example given in Figure 4.8a consists of two unambiguously defined rules \( A \rightarrow B' \) and \( A \rightarrow B \) and one derived communication rule \( A \xrightarrow{1} B \) which is redundant to the definition of the former rule. Having ambiguous rules is undesirable as they potentially hide the reason why a communication is allowed, still they do not corrupt the result of an analysis (as long as they carry the same permission). If the disallowing communication rules introduced in the next section are used, rules can not only be ambiguous but contradictory (see section 4.5). Such contradictions must be handled in order to prevent corruption of the analysis.
4.5. Disallowing Communication Rules (_refl)

On one hand communication rules may be used to solely allow communication while all communication not matched by communication rules is denied. On the other hand, communication rules may carry a permission to explicitly allow or deny matching communication. The later rules which deny communication are named disallowing communication rules. However communication rules carrying permissions do not increase the power of communication rules, that is, each architecture which can be modeled using communication rules carrying an explicit permission can also be modeled using only allowing communication rules. Nevertheless supporting communication rules with explicit permissions can significantly lessen the modeling effort as shown by an example later on. Disallowing communication rules are symbolized as $A \not\rightarrow B$ with the meaning that $A$ must not communicate with $B$. Disallowing communication rules are depicted similar to allowing ones as thick, but red colored arrows like shown in Figure 4.9. Furthermore disallowing communication rules can be derived in the same way as defined in section 4.4 and are symbolized accordingly. Derived disallowing communication rules are depicted similarly to derived allowing rules as arrows with dashed outlines but in pale red color. The following situation should exemplify the lesser modeling effort gained by using allowing and disallowing communication rules.

The situation encompasses six classes like the introductory example which are not part of a layered architecture but of a different architecture containing a facade. Class B1 should act as a facade to C1 and C2, while A1, A2 and A3 are clients of B1. This can easily be modeled using six disallowing communication rules as shown in Figure 4.10a. The same can be expressed without disallowing rules but with a higher modeling effort as it requires rules between all architectural entities except of the disallowing rules resulting in a total of 14 rules (as reflexive allowing rules can be omitted as explained later) like shown in Figure 4.10b.
Contradictory Rules

While the effort to model an architecture can be decreased by using disallowing communication rules, supporting these comes at a price as they introduce the possibility of contradictory rules: Two rules are contradictory if they apply to the same entities and their granted permission is different. In this sense contradictory rules are ambiguous rules but with conflicting permissions. A simplistic example of two contradictory rules is given in Figure 4.11a. Such a definition of rules is invalid and should be handled as an erroneous input. However together with the include relation described

Figure 4.10.: Example facade

(a) Realized by disallowing rules  
(b) Realized by allowing rules

Figure 4.11.: Contradictory rules

(a) Unintentional contradictory rules  
(b) Intentional contradictory rules
4.6. Resolving the Ambiguity of Rules

In order to resolve both ambiguous and contradictory rules some kind of priority that clarifies the precedence between rules is needed. This thesis takes an approach to use the number of includes needed to derive a matching communication rule from a defined one as the measure to determine the rule’s priority. This measure was already introduced in section 4.4 as the derivation number of derived rules. In the following it is described how the derivation number can be used to resolve the ambiguity of rules.

Consider a modeled architecture together with one allowing communication rule (green) between an entity A’ and entity B’ and one denying communication rule (red) between the entities A and B which are included by A’ and B’, respectively, as depicted in Figure 4.12a.

The allowing communication rule also yields three derived communication rules (pale green)\(^{10}\). Analyzing the rules discovers the following derivation numbers: Rule 1) is not derived and thus has a derivation number of 0. Rule 2) has a derivation number of 1 as it is derived from \( A' \rightarrow B' \) using the (direct) include relation \( A \in A' \). Rule 3) has a derivation number of 1 as well as it derived using the include relation \( B \in B' \). Rule 4) has a derivation number of 2 as it is derived using both of the former mentioned include relations. Rule 5) has as rule 1) a derivation number of 0 as it is not derived at all. Now analyzing the rules regarding their respective caller and callee entity results in

\(^{10}\)It is impossible to derive any further rules from the denying one, as neither caller nor callee of the rule include other architectural entities
the detection of rule 4) and 5) being ambiguous (as they both match communications between entity A and B) and are even contradictory (as their operators are different). Using the derivation number as means of priority in the sense that smaller values have higher priority (with 0 being the highest priority) allows for determination the precedence of rule 4) and 5) in the following way: Communication is denied because rule 5) with priority 0 overrules 4) with priority 2.

Using the derivation number as the priority cannot guarantee to resolve all possible ambiguous and contradictory rules quite the contrary it is easy to define two rules with the same caller and callee and different operators as seen in Figure 4.11a. However these definitions can be considered as erroneous inputs. Unfortunately it is also possible to encounter situations like depicted in Figure 4.12b. Rule 1) and rule 2) are not ambiguous as their caller and callee entities are different. Rule 3) and 4) share the same entities as caller and callee entity. In addition rule 3) and rule 4) have a derivation number of 1 as both are derived using the direct include relations $B \in B'$ respectively $A \in A'$.

Such situations cannot be resolved automatically but must be handled by the creator of the rules. Still, having priorities enables the definition of rules which seem to be contradictory but are not (taking into account their priority) and in turn allows for smaller and more elegant modeling of the communication rules as shown in the facade example in Figure 4.11b.

4.7. Software Architecture Metamodel

Four different concepts were introduced and analyzed in this chapter. More specifically two types of entities and two types or relations between them were described: Architectural entities, code-level entities, include relations and communication rules. The metamodel depicted in Figure 4.13 was derived from the former analysis. Both entity types, code-level entities (called SoftwareUnits) and architectural entities (called ArchitectureUnits), share the same base type Unit. Still only architectural entities may embed other entities forming the include relationship. Also the communication rule (called Rule) forms a relation exclusively between architectural entities. It may contain an additional permission if is desirable to explicitly deny communications between entities.

4.8. The Need for Default Rules

If each and every code-level building block of the subject software system is mapped to an architectural entity and if all possible communications are explicitly allowed or denied by communication rules there would be no need for default values. However explicitly defining all communication rules is not only tedious work, it also results in

---

They both are not derived and thus have a derivation number of 0 but a precedence between them is not needed.
4.8. The Need for Default Rules

A huge amount of rules, as for each architectural entity communication rules to every architecture entity (excluding the entity itself) must be defined. That leads to

\[ \sum_{i=1}^{n} (n - 1) = n \times (n - 1) > (n - 1)^2 \]

for \( n \) architecture entities. Such a quadratically increasing number of rules cannot be handled for real life software systems. For that reason omitting rules for the sake of simplicity is not only allowed but promoted by the concept of this thesis. In addition to omitted rules architectural or code-level entities might be missing. That is code-level entities might not be included by an architectural entity or an entity of the source code is not mapped to a code-level entity. While this is not desirable it should be possible to cope with such an incomplete input. In the following the three types of missing entities or rules are presented and the default behavior is described.

**Missing Code-Level Entities**

The actual communication inside a software system happens between a caller (the calling function/method) and its callee (the function/method being called). If one (or both) of these are not mapped to a code level entity no rule can be defined (and therefore no rule can be found) to assess the communication. In order to solve missing code-level entities a (global) default permission (with the value ALLOW or DENY) exists.

**Missing Architectural Entities**

Caller or callee are mapped to code-level entities, but at least one of them is not included by an architectural entity. As rules can only be defined between architectural entities, no rules affecting the not included code-level entity can be defined. In order to solve missing architectural entities another (global) default permission (with the value ALLOW or
DENY) exists.

**Missing Rules**

Caller and callee are mapped to different code level entities and those are included by an architectural entity. Still no rule was defined between any entity (directly or indirectly) including the caller and any entity including the callee. This is quite common when modeling a software architecture as omitting connections between entities usually means both entities should not interact. In order to solve missing rules yet another (global) default permission (with the value ALLOW or DENY) exists\(^\text{12}\).

### 4.9. Implicit Communication Rules

Apart from the default rules above whose permission are configurable there are two implicit rules which always allow certain types of communication. Both rules describe how communication within an entity is handled but are distinguished by the type of the entity (code-level or architectural).

**Same Code-Level Entity**

Code-level entities are considered atomic on the modeling level in the scope of this thesis as described in section 4.1. That makes it impossible to define any rule between parts of the same code-level entity. Therefore communication inside a code-level entity is always allowed.

**Same Architectural Entity**

In contrast to code-level entities are architectural entities not atomic. Still each code-level entity is directly included by (at most) one architectural entity. Communication within an architectural entity is therefore communication between two (different) code-level entities which are directly included by the same architectural entity. While it would be possible to define a communication rule that denies communication within an architectural entity it seems awkward from the modeling point of view to deny the communication within the whole entity (consider disallowing communication within a layer of the simple example from the introduction). Therefore communication between code-level entities directly included by the same architectural entity is always allowed, that is, there always exists an implicit rule \( A \rightarrow A \) for each architectural entity \( A \).

**Note:** Communication between indirectly included code-level entities is not implicitly allowed.

---

\(^{12}\) It might be useful to switch the value to DENY in order to find missing rules or to ALLOW in order to only check the already defined rules.
5. Realization

With realization of one’s own potential and self-confidence in one’s ability, one can build a better world.

Dalai Lama

Contents

<table>
<thead>
<tr>
<th>5.1. CIC Class Library</th>
<th>41</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2. Validation</td>
<td>53</td>
</tr>
<tr>
<td>5.3. Tools</td>
<td>62</td>
</tr>
</tbody>
</table>

According to the results presented in the last chapter a software solution named CommunicationIntegrityChecker (CIC) was developed. The software solution consists of a class library and two different applications based on the library. The class library implements the overall concept presented in the previous chapter along with additional features (like XMPP network communication) and necessary prerequisites (like imports of architecture information). The two applications MonitoringTool and InformationProcessor divide the task of monitoring and analyzing the communication of the subject software system. First the CIC class library shall be described. It consists of a couple of packages and more than one hundred classes and interfaces, some of them rather small (e.g. exception classes or observer interfaces and implementations) other more complex realizing concepts and features introduced in chapter 4.

5.1. CIC Class Library

The CIC class library consists—as previously mentioned—of more than one hundred classes making it not feasible to draw all of them as one big class diagram. Instead Figure 5.1 displays a simplified UML class diagram consisting of the most important packages and their classes and relations. The classes of the essential packages (architecture, rules, monitoring) shown in the simplified class diagram will be described in detail as follows: This class diagram features the UML dependency stereotypes «create» and «use» which are both predefined by the UML standard. A UML dependency is a “relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation”. This relationship is directed and points from the dependent element, called client, to the element the client depends on, called supplier. Stereotypes on dependencies are used to specify the manner of the dependency used. The stereotype «create» has the predefined
5. Realization

Figure 5.1.: Simplified class diagram of the CIC class library (UML)

semantic to “[denote] that the client classifier creates instances of the supplier”. The stereotype «use» specifies that “one element requires another element (or set of elements) for its full implementation or operation”. Apart from the predefined stereotypes, Figure 5.1 also makes use of the possibility to define individual stereotypes and uses the stereotypes «enhance» and «validate» which will be described later in subsection 5.1.2 and subsection 5.1.4, respectively. The packages are described with the functionality they provide, design details (e.g. used design patterns) of interest, the features the classes offer and their dependencies on other packages. Import and export of the architectural data (modeled by the packages architecture and rules) provided by the respective packages will be illustrated afterwards. CIC allows to publish the results of both its monitoring and its validation through XMPP. Results are serialized in the JSON format and send to an XMPP chat room to allow other programs for visualization and further analysis. This functionality is implemented in the packages network.xmpp and marshalling.

5.1.1. Package: rules

The abstract Rule class and its subclasses represent the communication rules defined in section 4.4. A rule always has an operator of type RuleOperator which corresponds to the Permission type with its two possible values Allowed and Denied. Furthermore it contains a priority attribute in order to assess the precedence of rules. The default priority value of rules passed as input (cf. subsection 5.1.6) is 0. It is possible for the user to specify rules with higher or lower priorities than 0, this however requires knowledge on how priority values of derived rules (cf. section 4.4) are calculated and is generally not advised. The DirectedBiCommunicationRule class as shown in Figure 5.1 implements the general $A \rightarrow B$ relation that is $A$ may communicate with $B$. Therefore it aggregates

\footnote{In order to maintain readability this package was omitted in the simplified class diagram.}
5.1. CIC Class Library

Figure 5.2.: Detailed UML class diagram of package rules

one ArchitectureUnit as the caller (left) and one ArchitectureUnit as the callee (right) part of the rule. The DirectedBiCommunicationRule class suffices to define all possible $A \rightarrow B$ rules as described in section 4.4. Nevertheless implementing special types for $* \rightarrow B$ and $A \rightarrow *$ (that is any ArchitectureUnit may communicate with $B$ and $A$ may communicate with any ArchitectureUnit) allows for a more convenient definition of rules. Furthermore SameArchitectureUnitRule and SameSoftwareUnitRule are used to realize the implicit rules from section 4.9. Dependencies: Most of the concrete Rule subclasses have dependencies to the ArchitectureUnit class of the architecture package.
5. Realization

Figure 5.3.: Detailed UML class diagram of package architecture

5.1.2. Package: architecture

The concepts of architecture and code-level entities introduced in section 4.2 and section 4.1 are realized as concrete classes ArchitectureUnit and SoftwareUnit respectively. Both classes share the same abstract base class Unit. These three classes were designed according to the Composite pattern [GHJV95] to realize the include relation as described in section 4.3. The abstract Unit class contains a name attribute which serves as the identifier for an instance of Unit. It also holds a reference on its parent to allow upwards navigating in its include tree\(^2\). The ArchitectureUnit class facilitates modeling an architecture by providing a named entity on the architectural level which can include other entities. Therefore it allows for aggregating other instances of the Unit class to implement the Composite pattern mentioned before and to realize the include relation. By this it allows downward navigation in its include tree. The SoftwareUnit class permits establishing a mapping between code-level entities (e.g. packages or classes) and callable entities (most likely methods or functions). This is supported by subclasses of the abstract SoftwareUnitFilter class which are called filters and allow to match a SoftwareUnit against a monitored call of a callable entity. For example the ClassSoftwareUnitFilter class is used to successfully match a SoftwareUnit against all monitored calls of its methods. Other filters allow matching against a package or a regular expression. Lastly the architecture package contains a class which offers the ability to map a recorded method execution to the SoftwareUnit it belongs to. This functionality is called enhancing in the context of this thesis and the class providing this is therefore called Enhancer. Dependencies: The ExecutionRecordEnhancer class has a dependency to the ExecutionRecord class and the EnhancementState enumeration of the monitoring package.

\(^2\)This parent association was omitted in Figure 5.1 to maintain readability.
5.1.3. Package: monitoring

The monitoring package models execution traces of the subject software system for the rest of the CIC class library. An execution trace is a sequence of method calls providing the information which method was called and when. This is modeled by the ExecutionRecord class which holds information about the method invoked (attribute target), the point in time when the execution of the method started (attribute start) and ended (attribute end). In order to analyze an ExecutionRecord it first must be enhanced (cf. subsection 5.1.2), i.e. storing a reference to the SoftwareUnit the executed method belongs to. This allows not only access to the SoftwareUnit itself but to all including ArchitectureUnit upwards the include tree. In order to track if there was an attempt to enhance a specific ExecutionRecord and if so whether this attempt was successful or not the class manages a state (attribute state) supporting three values for the state: ENHANCEMENT_NOT_PERFORMED, ENHANCEMENT_UNSUCCESSFUL, ENHANCEMENT_SUCCESSFUL. An ExecutionRecord instance in state ENHANCEMENT_SUCCESSFUL is guaranteed to
5. Realization

hold a reference to its SoftwareUnit, while ExecutionRecords in the other two states never have this reference.

In the context of this thesis it is essential to have pairs of records of executed methods instead of have single records. Combining records to pairs of the calling method (short caller) and the method being called (short callee) provides exactly the form of communication within a subject software system we want to analyze and validate. This pairs of execution records are modeled by the class ExecutionRecordPair which aggregates two ExecutionRecord as caller and callee. Validation is described at full length in subsection 5.1.4. Still validating an ExecutionRecordPair instance always yields a ValidationResult which is stored inside the validated instance of ExecutionRecordPair. Similar to the ExecutionRecord class the ExecutionRecordPair class holds a state (attribute validationState). This state however does not track the enhancement of the records but the validation of record pairs. The type of the state attribute is contained in the package validation and is therefore described in detail later on but in general an instance of ExecutionRecordPair with a validation state of value VALIDATION_NOT_PERFORMED never holds a ValidationResult instance while it always does when having any other of the validation state values.

CIC does not execute the monitoring of the method calls and the conversion to pairs of methods calls itself. Instead it relies on functionality provided by monitoring frameworks. Currently it supports only the Kieker monitoring framework which was previously discussed in section 3.4. Kieker provides fast and light-weight monitoring and creation of message traces which can be used to extract the required pairs of execution records. Furthermore through AspectJ Kieker allows to weave monitoring of single methods into an already compiled Java application. This unobtrusive approach enables analyzing a subject software system without manipulating its source code. However CIC is extensible and it should be easy to adopt other monitoring frameworks as long as they provide the execution data and information to establish the execution record pairs. Dependencies: The ExecutionRecord class has a dependency to the SoftwareUnit of the architecture package. The ExecutionRecordPair has dependencies to ValidationState and ValidationResult of the validation package.

5.1.4. Package: validation

Validating the executed methods of a target software system is the prime objective of this thesis. Validation is implemented in package possessing the same name, which consists mainly of the classes Validator, ValidationStrategy, its subclasses and ValidationResult and the two enumerations Permission and ValidationState. The Validator class analyzes the communication information provided by an instance of ExecutionRecordPair of the previously described monitoring package. It then performs a series of checks to implement the behavior regarding the default values described in section 4.8 and the implicit rules described in section 4.9. Depending on the previous checks, that is whether a performing a validation is possible and is needed, the validation is then executed.
by delegating it to a ValidationStrategy instance. The checks performed and details about how the two classes implementing the ValidationStrategy will be described in detail in Section 5.2. Validating an execution trace always yields to an instance of ValidationResult. The UML stereotype «validate» is used to denote that a Validator instance supplies an ExecutionRecordPair instance with the result of its validation. Such a ValidationResult instance can hold up to two rules: The rule (attribute rule) which matched the caller/callee pair of the validated ExecutionRecordPair instance and another rule (attribute contradictoryRule) also matching the caller/callee but being contradictory to the first rule as defined in Section 4.5. Both references might be empty (e.g. a contradictory rule can only exist if at least one matching rule exists and that cannot be guaranteed). Most importantly ValidationResults carry a permission attribute which states whether the executed method call was allowed by the architectural information (code-level and architectural entities and the defined rules together with the implicit rules and default values) or not. This information is expressed through the enumeration Permission featuring the two values ALLOWED, DENIED. Lastly the ValidationResult incorporates a state attribute of type ValidationResult which exposes the cause for the evaluated permission. In general the ValidationResult enumeration states whether a validation was performed and if so whether the prerequisites for the validation were provided and finally states the result of the validation in a generic manner, that is independent of actually defined entities or rules. The values of ValidationResult and their consequences for a ValidationResult instance in this state are defined as follows:

**VALIDATION_NOT_PERFORMED**

No validation was performed. This value is never used as the state of a ValidationResult but is the initial value of an ExecutionRecordPair instance before validation is performed.
5. Realization

VALIDATION_FAILURE_MISSING_CALLER
The calling method was not monitored thus it is impossible to apply any rule (cf. section 4.8).
The Permission of the ValidationResult is set accordingly to the default value allowUnenhanced. Both attributes rule and contradictoryRule are null.

VALIDATION_FAILURE_MISSING_CALLEE
The method being called was not monitored thus it is impossible to apply any rule (cf. section 4.8).
The Permission of the ValidationResult is set accordingly to the default value allowUnenhanced. Both attributes rule and contradictoryRule are null.

VALIDATION_FAILURE_NO_SOFTWARE_UNIT
Both caller and callee were monitored, still at least one of them was not mapped to a code-level entity and thus it is impossible to apply any rule (cf. section 4.8).
The Permission of the ValidationResult is set accordingly to the default value allowUnenhanced. Both attributes rule and contradictoryRule are null.

VALIDATION_FAILURE_NO_ARCHITECTURE_UNIT
Both caller and callee were monitored and mapped to code-level entities, still at least one of the entities was not mapped to an architectural entity and thus it is impossible to apply any rule (cf. section 4.8).
The Permission of the ValidationResult is set accordingly to the default value allowUnenhanced. Both attributes rule and contradictoryRule are null.

VALIDATION_NO_MATCHING_RULE
Both caller and callee were monitored, mapped to code-level entities themselves mapped to architectural entities, however no rule matching these architectural entities was defined.
The Permission of the ValidationResult is set accordingly to the default value allowUnruled. Both attributes rule and contradictoryRule are null.

VALIDATION_MATCHING_RULE
Both caller and callee were monitored, mapped to code-level entities themselves mapped to architectural entities and at least one rule matching these architectural entities was defined (cf. section 4.4, section 4.5).
The Permission of the ValidationResult is set accordingly to the operator of the matched rule. The attribute rule holds a reference to the matched rule while the attribute contradictoryRule is null.

VALIDATION_CONTRADICTORY_RULES
Both caller and callee were monitored, mapped to code-level entities themselves mapped to architectural entities and at least two rule matching these architectural entities was defined. However at least two of the rules are contradictory as described in section 4.5.
The Permission of the ValidationResult is set accordingly to the default value allowUnruled.
allowContradictoryRules. The attribute rule holds a reference to the matched rule while the attribute contradictoryRule holds a reference to a rule contradicting the matched rule.

VALIDATION_SAME_SOFTWARE_UNIT
The caller and callee belong the same code-level entity and are therefore always allowed (cf. section 4.9).
The Permission of the ValidationResult is set to ALLOWED. Both attributes rule and contradictoryRule are null.

VALIDATION_SAME_ARCHITECTURE_UNIT
The caller and callee belong the same architectural entity and are therefore always allowed (cf. section 4.9).
The Permission of the ValidationResult is set to ALLOWED. Both attributes rule and contradictoryRule are null.

Dependencies: The validation package has a dependency on the ExecutionRecordPair from the monitoring package as it validates instances of this class and assigns the result of a validation to the ExecutionRecordPair object in question. In order to perform the validation it uses both the defined architectural (ArchitectureUnit) and code-level (SoftwareUnit) entities through a UnitCatalog instance. All these classes are from the architecture package. It further analyzes the communication by applying the rules it retrieves from a RuleCatalog instance. The different rule types and the RuleCatalog are part of the rule package.

5.1.5. Packages: network.xmpp & marshalling

Together the network.xmpp package and the marshalling package provide functionality to publish information (in particular execution records with their enhancement and validation information) in a XMPP chat room. The former package allows connecting to chat rooms and sending messages to them and receiving messages from them. The latter package conducts the required marshalling from objects to a serialized representation and unmarshalling a serialized representation back to an object in order to send or receive objects. Figure 5.7 shows a screenshot of a published execution record received by an XMPP client. XMPP (Extensible Messaging and Presence Protocol) is “a set of open technologies for instant messaging, presence, multi-party chat, voice and video calls, collaboration, lightweight middleware, content syndication, and generalized routing of XML data” [XMP14]. XMPP is (like many internet technologies) based on the client-server model. It allows multiple clients to communicate through a mutual server. The XmppCommunicator class which is the central class of the network.xmpp package allows to connect to a XMPP chat room and when connected send and receive messages to, respectively from, the chat room. The XMPP communication is based
5. Realization

on Smack an “Open Source XMPP (Jabber) client library” \[Ign14a\] which implements the XMPP message protocol \[SATII\]. The XMPP message protocol itself is XML-based and commonly transfers textual payloads. It however is extensible and allows custom extensions to the protocol.

For the CIC communication the protocol was slightly extended using properties \[Ign14b\] to allow distinguishing between different types of the transferred objects. The payload itself however was chosen to be textual serialization of the objects in the JSON (JavaScript Object Notation \[Cro06, jso14\]) format. JSON was selected as the serialization format because it is an open standard with extensive tool support which allows easy integration of other applications as consumer or producer of serialized objects in the scope of CIC. The marshalling itself is defined through a small and simple interface and implemented inside the marshalling package. The JSON-marshalling is based on the google-gson \[gso14\] library which provides simple conversion between POJO (plain old Java objects) and JSON representations \[SLW11\]. In order to enable this conversion the marshalling package contains flat versions of the ExecutionRecord and the ExecutionRecordPair class. These classes named FlatExecutionRecord and FlatExecutionRecordPair are designed to hold copies of the original instances but with their references to other instances like rules or entities replaced by string representations. While CIC currently only offers marshalling in the JSON format it is designed to be extended in the future by new marshellers implementing the interface if needed. Dependencies: The marshalling and the network.xmpp package have dependencies on the classes they marshal or send. In detail they depend on the ExecutionRecord, ExecutionRecordPair classes and all subclasses of the Unit and Rule class.

5.1.6. Packages: importer & exporter

The importer and exporter package realizes the import of existing architectural information and rule definitions and their export, respectively. The XmlArchitectureImporter allows to read an XML file containing both architectural and code-level entities. As previously said these entities form a forest of trees. Each tree has

![Figure 5.7.: Execution record pair (XMPP message)](image-url)
an architectural entity as its root, again architectural entities as inner nodes and finally
code-level entities as leafs. The forest of tree structure of the entities is combined with
the ability of XML to directly allow nested entries in order to permit well-arranged entity
definitions. In detail it allows defining the root entities together with their children
and further descendant without the need of referencing entities. Listing 5.1 shows
the definition of the first layer 'L1' of the architecture in the introductory example.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<architecture>
  <architectureUnit name="L1">
    <units>
      <softwareUnit name="SU1">
        <filter>
          <classFilter>org.example.samplecom.Unit1</classFilter>
        </filter>
      </softwareUnit>
      <softwareUnit name="SU2">
        <filter>
          <classFilter>org.example.samplecom.Unit2</classFilter>
        </filter>
      </softwareUnit>
      <softwareUnit name="SU3">
        <filter>
          <classFilter>org.example.samplecom.Unit3</classFilter>
        </filter>
      </softwareUnit>
    </units>
  </architectureUnit>
</architecture>
```

Source Code 5.1: Architecture import (XML definition)

The remaining architectural entities L2 and L3 as well as their included code-level
entities are defined similarly and are here omitted for brevity. The architecture node is
the XML root node required by the XML format. The nodes of type architectureUnit
and softwareUnit represent instances of ArchitectureUnit and SoftwareUnit. Both
require an name attribute whose value has to be unique to distinctly identify the entities.
A softwareUnit node must have exactly one filter child node, which determines how the
SoftwareUnit instance matches against monitored communications (cf. subsection 5.1.2).
An architectureUnit on the other hand must have a units node containing a list of
included entities of type architectureUnit or softwareUnit\(^3\). The format of an XML
architecture definition is enforced by an XML Schema Definition (included in the
appendix if this thesis) which is used to validate the definition before parsing it. For

\(^3\)As the example architecture does not contain architectural entities included by other architectural
elements all children in the XML file are of type softwareUnit.
5. Realization

the sake of completeness an XmlArchitectureExporter allowing to export in-memory architecture definitions is provided as well.

After importing the architectural information communication rules can be imported. As previously said a communication rule always has a priority and a rule operator, which causes to allow or deny an actual communication when validated. As seen in the definition of the first directedBiCommunicationRule entry priority and operator can both be set through the attributes “priority” and “permission”, respectively. They, however, can both be omitted yielding to their default values 0 and ALLOWED as in all other entries. The different types of rules was already discussed in subsection 5.1.1 In general the possible types of rules in an XML rules definition correspond to the class names of the different rule types with their first character in lower case. Listing 5.2 shows the definition of all rules in the introductory example.

1. 

Source Code 5.2: Rules import (XML definition)

The rules node is the root node requested by XML and has no significance to the rule definition itself. Its children are ruleSet nodes which feature a name and aggregate a list of different rule nodes. The ruleSet nodes allow to organize rules (e.g. to hint from which document the rules were retrieved) but have no semantic meaning to the validation. The ruleSet node named “default” is required and shall remind the user that the two rules SameSoftwareUnitRule and SameArchitectureUnitRule are present in every rules definition. The rules which are defined a child nodes of a ruleSet node have one of the types directedBiCommunicationRule, callerCommunicationRule or calleeCommunicationRule. A directedBiCommunicationRule entry represents a rule between two previously defined architectural entities (commonly imported by the
5.2. Validation

XmplArchitectureImporter introduced above) and as such must have both a caller and a callee child node, which reference the corresponding entities through their name attribute. A callerCommunicationRule or calleeCommunicationRule entry represent a kind of wildcard rule in which only the caller or callee are defined and communication from or to this entities are generally allowed (or denied depending on the actual permission of the rule). Dependencies: The ex- and importer have dependencies to the classes whose instances they create from an XML definition or whose instances they serialize as XML.

5.2. Validation

The validation of the communication integrity is the prime objective of this thesis. The general design of the validation package was already described in subsection 5.1.4. In this section however the validation algorithm is explored in great detail. As mentioned before the Validator class uses a (validation) strategy to perform the actual validation. In order to ensure the implicit rules (cf. Section 4.9) and the default values for incomplete architectural information (cf. section 4.8) it also implements the template method pattern in a simplistic way to provide a generic algorithm which evaluates all information except the actual rules as shown in Source Code 5.3. In detail the algorithm first checks whether both caller and callee where monitored.
private ValidationResult executeValidation(CallerCalleePair pair) throws ValidationException {
    ValidationResult result = null;
    ExecutionRecord callerRecord = pair.getCaller();
    ExecutionRecord calleeRecord = pair.getCallee();
    SoftwareUnit callee = calleeRecord != null ? calleeRecord.getSoftwareUnit() : null;
    SoftwareUnit caller = callerRecord != null ? callerRecord.getSoftwareUnit() : null;

    // validation: caller not monitored
    if (callerRecord == null) {
        result = tool.handleMissingCaller();
    }
    // validation: callee not monitored
    else if (calleeRecord == null) {
        result = tool.handleMissingCallee();
    }
    // validation: no software unit
    else if (caller == null || callee == null) {
        result = tool.handleUnhancedMissingSoftwareUnit();
    }
    // validation: same software unit
    else if (caller == callee) {
        result = tool.handleSameSoftwareUnit();
    }
    // validation: no architecture unit
    else if (caller.getParentUnit() == null || callee.getParentUnit() == null) {
        result = tool.handleUnhancedMissingArchitectureUnit();
    }
    // validation: same architecture unit
    else if (caller.getParentUnit().equals(callee.getParentUnit())) {
        result = tool.handleSameArchitectureUnit();
    }
    // validation: two different architecture units from two different software units
    else {
        List<Rule> matchingRules = validationStrategy.validate(
            caller.getParentUnit(), callee.getParentUnit());
        result = tool.handleValidationNeeded(matchingRules);
    }
    log.info(String.format("Validation result between %s and %s: %s", caller, callee, result));
    return result;
}

Source Code 5.3: Validator.validate

Then it checks that the monitored caller/callee pair was mapped to code-level entities.
Then it enforces the implicit rule that calls within a code-level entity are always allowed. Afterwards it checks whether both caller and callee were mapped to architectural entities and enforces the implicit rule that calls within an architectural entity are always allowed. Only if the precondition of having two different architectural entities is met that ValidationStrategy instance is to discover all matching rules.

That approach allows all concrete ValidationStrategy implementations to consider getting two valid instances of ArchitectureUnit a precondition and freeing them from realizing any logic besides finding matching rules of the two ArchitectureUnit instances. Even more a ValidationStrategy does not need to handle contradictory rules. Instead each strategy is asked to return all matching rules of the same minimal priority. Contradictions are handled by the Validator class and its helper class ValidatorTool. That eases the effort implementing subclasses of ValidationStrategy and reduces their complexity. It also ensures that the default values and implicit rules are always computed in a correct manner and contradictory are always treated properly independent from the strategy used.

5.2.1. ValidationStrategies

In the following the two implementations of the ValidationStrategy featured by CIC are presented. They differ in the approaches taken as well as in the used technology. The SimpleValidationStrategy which relies only on the Java programming language and classes provided by CIC shall be described first. It utilizes the concept of derived communication rules implicitly without implementing or instantiating any derived rules. In contrast to the SimpleValidationStrategy the second strategy called DroolsValidationStrategy utilizes the rule engine “Drools Expert” which is the basis of the business rule management system “JBoss Enterprise BRMS”. It also makes direct use of derived communication rules by (automatically) deriving them from the defined rules.

5.2.2. SimpleValidationStrategy

The SimpleValidationStrategy implements its abstract ValidationStrategy base class and provides the requested validation feature through comparison of the defined rules with the given caller/callee entity pair (and their including entities as explained soon). As mentioned before this strategy does not directly use derived communication rules i.e it does not instantiate new communication rules. Instead it calculates the distances of the caller and callee entity to their (directly and indirectly) including entities for which (not derived) rules are defined. It than uses the sum of the caller and callee distances as the priorities of these rules. This approach utilizes the definition of derived communication rules from section 4.4:

\[
\begin{align*}
x^k \rightarrow y & \iff \exists x' \exists y' \exists i \exists j : (i + j = k \land x \in^i x' \land y \in^j y' \land x' \rightarrow y')
\end{align*}
\]
As depicted in Figure 5.8 finding all rules matching two given architectural entities (caller and callee) is easy. It suffices to use three nested loops iterating 1) through all architectural entities which include the caller (going upward the include tree of the caller), 2) through all architectural entities which include the callee (going upward the include tree of the callee), 3) through all rules and finally comparing both caller and callee with the caller and callee of each rule. This however finds all matching rules, not only matching rules sharing the highest priority which requires to remove rules having lower priorities afterwards. This approach is also suboptimal in its run-time behavior as it requires \(d_{\text{caller}} \times d_{\text{callee}} \times |\text{rules}|\) comparisons (with \(d_{\text{caller}}\) and \(d_{\text{callee}}\) being the the distance of the caller and callee in the include tree and \(|\text{rules}|\) the number of all defined rules). The desired result of a validation strategy however is not finding all matching rules but all matching rules with the same, highest priority (that is the lowest numeric priority value). If the highest priority of the matching rules were known beforehand the needed comparisons to identify those rules would be narrowed to only the rules having this priority. Unfortunately the highest priority is not known before the start of the validation. However the priorities of the rules can be utilized to speed up the algorithm by analyzing the rules in ascending order of their priorities values. This enables to analyze rules with higher priorities before rules with lower ones. Using this approach the first matching rule found determines the highest priority. The algorithm then needs to find the remaining rules with the same priority but does not need to search for other rules with lower priorities. Source Code 5.4 shows the implementation of this algorithm. The algorithm starts with fetching all including entities of both the caller and callee entity to be validated. The lowest possible priority which is numerically the highest value is the sum of the include levels of the caller and callee. This value is calculated next and stored as the maxPriority variable which restricts the loop variable of following loop. The loop implements the descending order of rules based on their priority (or ascending based on their priority value). For each priority to be analyzed...
private List<Rule> findAllMatchingRules(ArchitectureUnit caller, ArchitectureUnit callee) throws UnknownRuleType {
    List<Rule> matches = new ArrayList<>();
    Map<Integer, ArchitectureUnit> callers = caller.findAllAncestorsWithDistance();
    Map<Integer, ArchitectureUnit> callees = callee.findAllAncestorsWithDistance();
    int maxPriority = callers.size() + callees.size();

    // also check direct caller and callee (they have distance 0)
    callers.put(0, caller);
    callees.put(0, callee);

    for (int priority = 0; priority <= maxPriority; ++priority) {
        // find all addends with i+j=priority
        for (int i = 0; i <= priority; ++i) {
            caller = callers.get(i);
            callee = callees.get(priority - i);
            if (caller != null && callee != null) {
                for (Rule rule : findDirectMatchingRules(caller, callee)) {
                    Rule derivedRule = rule.clone();
                    derivedRule.setPriority(rule.getPriority() + priority);
                    matches.add(derivedRule);
                }
                if (!matches.isEmpty()) {
                    break;
                }
            }
        }
    }

    return matches;
}

Source Code 5.4: SimpleValidationStrategy.findAllMatchingRules
a second loop is used to find all pairs of entities whose parts include the caller/callee pair to be validated and the sum of their distance equals the desired priority. That is all entity pairs that hold: \((x, y): \text{caller} \in^i x \land \text{callee} \in^j y \land i + j = k\)

These entity pairs are candidates for callers and callees of existing rules. These candidates are then compared by findDirectMatchingRules against all actually existing rules which returns a list of rules matching the candidates. This list might be empty but if it is not the containing rules are added to the validation result. After all candidates were checked the algorithm ends if at least one rule was found or the lowest priority was already reached (\(priority = max\text{Priority}\)). Otherwise it continues to check the next lower priority. In the end the result might be empty (no matching rules), contain exactly one (one matching rule determining the validation result) or more than one matching rule (ambiguous, possibly contradictory rules). As described in section 5.2 it falls to the validator itself to evaluate the result.

This approach is superior to the first approach as it finds only rules having the highest priority. That eliminates the need to remove low priority rules, which are not to be evaluated for the validation and in general causes less comparisons. However in the worst case—that is no matching rules can be found—the number of comparisons is the same as in the first approach. The effort of analyzing the rules can be further decreased by precomputing the derived communication rules. That idea is implemented in the second validation strategy described next.

5.2.3. DroolsValidationStrategy

The DroolsValidationStrategy implements its abstract ValidationStrategy base class and provides the requested validation feature using the rule engine “Drools Expert” [JB14b] provided by JBoss [JBo14], a division of Red Hat, Inc. Before going into details about how the validation strategy performs its validation feature a minimalistic introduction to the Drools Expert rule engine is given.

The Drools Expert rule engine consists of the “Inference Engine”, rules build of a condition and an actions part and facts that are arbitrary Java Objects (or Plain Old Java Object, shortly POJOs). The conditions of the rules are matched by the “Pattern Matcher” inside the Inference Engine like depicted in Figure 5.9. The rules can be defined using Drools native rule language. A Drools rule must be written according to the schema given in Source Code 5.5. A rule name is a simple string to identify the rule. The condition part of the rule determines a set of facts...
5.2. Validation

Source Code 5.5: Schema of drools rules

\[
\begin{align*}
\text{rule } & \text{<ruleName> when } \\
& \text{<conditions> then } \\
& \text{<actions>};
\end{align*}
\]

that are to be matched by the rule. Lastly the actions are Java statements which are executed for set of facts matched by the rule. A condition may bind a variable to a matched fact and such a bound variable might be used further both in the condition and the actions part of a rule. The rule formulated in Source Code 5.6 which originates from the Drools Expert user guide [JB14c] gives an example of a simple rule:

Source Code 5.6: A simple underage check [JB14a]

The condition matches all facts that are objects of type Applicant and have an age attribute whose value is smaller than 18. The action of the rule calls the method setValid() with the boolean value false for all those objects in order “to disqualify any applicant younger than 18”.

The ValidationStrategy base class requires to implement its abstract method validate which accepts two parameters caller and callee both of type ArchitectureUnit. It is requested to return a list of matching rules of the highest priority. In order to provide the expected functionality the DroolsValidationStrategy runs a rule engine with different rules accomplishing one of the following tasks:

- Derive communication rules from existing ones.
- Remove ambiguous communication rules.
- Check communication rules for contradictions as a sanity check for the input.
- Validate an actual communication against the given and derived communication rules.

Furthermore the rule engine contains the communication rules as facts. It should be stressed here, that the rule engine contains (Drools) rules which must not be confused with the communication rules which instead are inserted as facts like previously said.

\footnote{The Applicant class needs to have an age attribute that is public or accessor methods obeying to the JavaBeans naming convention.}
5. Realization

```
rule 'derived left rules' when
   // handle one rule
   $rule : DirectedBiCommunicationRule(
      $caller:= caller, $callee:= callee)
   // find a child unit of the caller
   $child : ArchitectureUnit($caller:= parentUnit)
then
   // insert the new derived rule
   DirectedBiCommunicationRule newRule =
      new DirectedBiCommunicationRule(
         $child, $callee,
         $rule.getOperator(), $rule.getPriority()+1);
   insert(newRule);
end

// same as 'derived left rules' for the callee
rule 'derived right rules' when
   $rule : DirectedBiCommunicationRule(
      $caller:= caller, $callee:= callee)
   $child : ArchitectureUnit($callee:= parentUnit)
then
   DirectedBiCommunicationRule newRule =
      new DirectedBiCommunicationRule(
         $caller, $child,
         $rule.getOperator(), $rule.getPriority()+1);
   insert(newRule);
end
```

Source Code 5.7: DroolsValidationStrategy derive_rules.drl

The architectural entities (objects of type ArchitectureUnit) are inserted into the rule engine as facts as well. When all facts are loaded the Drools rules “derived left rules” and “derived right rules” are evaluated by the rules engine.

Each of the two Drools rules derive one communication rule at most from one existing communication rule. As both differ only in whether they derive the new communication rule from the caller or the callee part of a rule it suffices to explicate the first rule. The “derived left rules” contains two conditions:

- There must be a communication rule. The caller and callee parts of the communication rule are bound to the variables $caller and $callee.

- There must be an architectural entity which is included by the previously bound caller part of the communication rule.
5.2. Validation

// remove derived directed rules with lower priority
// (that is higher numerical value)

rule *remove duplicated directed rules* when
  $rule: DirectedBiCommunicationRule(
    $caller:=caller, $callee:=callee,
    $operator:=operator, $minPriority:=priority)
  $lowerPriorityRule: DirectedBiCommunicationRule(
    this != $rule, caller==$caller, callee==$callee, operator==$operator,
    priority >=$minPriority)
then
  retract ($lowerPriorityRule);
end

Source Code 5.8: DroolsValidationStrategy remove_ambiguous_rules.drl

The action part of the Drools rule consists only of the construction of the new, derived communication rule and the insertion of it as a fact. The Drools rule accomplishes to derive a communication rule utilizing the inductive rule for the left side—that is the caller—of an existing communication rule: \( x \xrightarrow{i+1,j} y \iff \exists x' : x \in x' \land x' \xrightarrow{i,j} y \) as described in detail in section 4.4. The same applies to the Drools rule “derived right rules” only that it derives from the right side—that is the callee—of an existing communication rule: \( x \xrightarrow{i,j+1} y \iff \exists y' : y \in y' \land x \xrightarrow{i,j} y' \). As the rules engine guarantees to match a rule against all matching facts—in particular to newly inserted ones—exactly once the two rules generate all possible derived rules (and no duplicates).

Some of the initially defined, not derived communication rules may be ambiguous. Even more deriving new communication rules may lead to new ambiguous rules as explained in section 4.4. The rule shown in Source Code 5.8 removes such ambiguous rules preserving only the one of the ambiguous rules with the highest priority. The Drools rule “remove duplicated directed rules” again contains two conditions:

- There must be a communication rule. All parts that is the caller, callee, operator and the priority parts of the communication rule are bound to variables.
- There must be a second communication rule which is different from the first, has the same caller, callee and operator and has a higher priority value (which means a lower priority).

The action part of the Drools rule consists only of removing the communication rule of the facts. It should be noted here, that only ambiguous and not contradictory rules are removed. As said before contradictory rules with the same priority cannot be resolved automatically and must therefore be resolved by the creator of the rules.

The last rule to apply is the actual validation which is simplified for validation strategies to only find all matching rules having the highest priority. The Drools rule
5. Realization

```
rule 'validation rule' when
    $comm : DirectedBiCommunication()
    $rule : DirectedBiCommunicationRule(
        caller==$comm.caller, callee==$comm.callee,
        $minPriority:=priority)
not DirectedBiCommunicationRule(
        caller==$comm.caller, callee==$comm.callee,
        priority<$minPriority)
then
    $comm.addMatchedRule($rule);
end

Source Code 5.9: DroolsValidationStrategy validation.drl
```

“validation rule’ contains three conditions:

- There must be an object of type DirectedBiCommunication. That object is created by the strategy itself and is used to retrieve the result from the rules engine back into the strategy. It is replaced by a new instance for the next validation by the strategy.

- There must be a communication rule. The priority, caller and callee parts of the communication rule are bound to variables.

- There must not be a second communication rule which has the same caller and callee and has a higher priority. That ensures that only communication rules sharing the same high priority are matched.

The action part of the Drools rule consists only of adding the matched rule to the result. The rules implementing the sanity check for the defined rules, that is to check for contradictory rules in the input and in the communication rules derived from these, is not depicted here for the sake of brevity. The approach however is similar to the presented Drools rules. In this case the rule can by summarized as: Find two rules having the same caller and callee, the same priority but different operators. The results are retrieved by the DroolsValidationStrategy in a similar way as in the Drools rule 'validation rule' presented before.

5.3. Tools

CIC as a realization of the presented concept consists of three main tasks: monitoring, enhancement and validation. The tasks must be executed in the given order and for each single pair of execution records. The functionality for each task is realized in the CIC class library.
Instead of simply assembling this functionality into one application which outputs its result into a file a more flexible approach was chosen. The three tasks are not programatically wired together but instead sequentially chained. The different classes, e.g. the Validator class, performing the tasks all implement the observer pattern [GHJV95] (as observerables) allowing other classes to get informed when a task is fulfilled. A class exists for each of the observerables which marshals and publishes the result of it into an XMPP chat room. That allows to distribute the various task over several applications and hosts. It furthermore allows to add different tasks into the chain easily. For example caches for the enhancement or the validation might simply be added at the appropriate positions of the chain. More importantly visualizations or further analysis of the validation data can be added at the end of chain.

In the current configuration, as shown in Figure 5.11, the monitoring is decoupled from the other two tasks while the enhancement and validation are coupled to increase the performance. The current configuration is realized through the two tools MonitoringTool and InformationProcessor. The MonitoringTool monitors the execution of a subject software system. It creates execution records for all monitored executions, bundles them as pairs of records consisting of a caller and a callee record. It finally serializes or marshals them and publishes them in an XMPP chat room. The InformationProcessor on the other hand waits for the marshaled record pairs. It unmarshals each pair, enhances both the pair’s caller and callee by setting their architectural information and than validates the pair. Finally it again marshals the pair and publishes the pair which now contains both architectural and validation information. Both tools read a properties file which contains configuration data like the XMPP user and chat room to use (both tools) or which validation strategy to choose (MonitoringTool only).
6. Evaluation

Everything that can be counted doesn’t necessarily count; everything that counts cannot necessarily be counted.

ALBERT EINSTEIN

6.1. MosAIC

In order to improve the software development process a multitude of best practices were developed. These best practices are aggregated as models and/or standards like CMMI-DEV, SPICE, V-Modell XT. In the context of MosAIC (Model supported Adoption and Assessment of Improvement Concepts) these collection are named Practice Repositories (short PR). While these PRs are meant to be adopted partly or as a whole by an organization it is as well possible to mix these by adopting combinations of parts of different PRs. MosAIC tries to ease this mixed adopting of multiple PRs which contain many similarities that are not obvious as each PR has its own nomenclature.

The MosAIC approach therefore provides metamodels in order to model the PRs in the same way and by this normalize them [JLR13 JOC+13]. Normalizing the PRs enables model analysis that is “computation of the similarity degree of two or more practices, the identification of practice dependencies, the categorization of practices according to their output, and the selection of best-suited practices are only some examples that can be implemented based on the framework” [JOC+13]. In summary MosAIC is a “systematic and extensible tool supported approach which can be used to facilitate an effective and efficient adoption of multiple PRs and of multiple PRs based assessment”. The mentioned tool support is realized by the MosAIC web application whose architectural overview is depicted in Figure 6.1.
6. Evaluation

6.2. Evaluation of the MosAIC Web Application

Following the approach taken in this thesis when validating a subject software system’s communication its semantics can be widely ignored. However the required input in order to perform the validation consists of both a log of monitored runtime execution and the subject software systems’ architecture, structure of its implementation and a mapping between them. While the former can be obtained relatively easy as described below the latter requires the architecture description and help from the software architect of the subject system. The person in charge of the ongoing development of the MosAIC web application kindly provided the architecture description and took her time to guide the creation of the mapping between the architecture and implementation of the subject software system, as well as, discussing the validation result. The provided architecture description consists of three illustrations which are part of a paper\(^1\) presenting the MosAIC approach. The high-level architecture of MosAIC is depicted in Figure 6.1.

\(^1\)The paper was not yet published by the time of writing this thesis but it was kindly provided by its author.

Figure 6.1.: MosAIC architecture overview
The MosAIC web application communicates with web browsers as its clients. It also exchanges data with a database server. Both clients and the database server can be neglected regarding the validation as they are not part of the architecture nor the implementation in a narrowed sense. Instead the MosAIC web application consisting of a web tier and a business tier is subject software system which is to be validated. The two tiers are candidates for the top-level architectural entities of the model of the subject software system which should allow the validation. The web tier is depicted in more detail in Figure 6.2. The business tier is divided into the two layers “Logic layer” and “Data layer” as depicted in Figure 6.3. It also reveals the name of the only layer of the web tier: “Presentation layer”. The layers were chosen to form as the architectural entities of the highest level of the architecture model created for the validation. By this the tiers are ignored as they do not provide any information—as showing additional includes or indicating communication rules—in comparison to the three layers. The components of
6. Evaluation

The layers as depicted in the Figure 6.2 and Figure 6.3 have a lesser degree of abstraction and already indicate communication rules. Performing the evaluation required several steps:

1. Preparation of monitoring of the MosAIC web application and execution of a user function of the MosAIC application in order to monitor the internal communication of the software system.

2. Creation of architectural entities and their include relations derived from the
6.2. Evaluation of the MosAIC Web Application

<table>
<thead>
<tr>
<th>Architectural entity</th>
<th>Included architectural entities</th>
</tr>
</thead>
<tbody>
<tr>
<td>PresentationLayer</td>
<td>{ModelingFacelets, AnalysisFacelet, ModelingPresenters, ModellingControllers, AnalysisPresenters}</td>
</tr>
<tr>
<td>BackingBeans</td>
<td>{}</td>
</tr>
<tr>
<td>LogicLayer</td>
<td>{Facades, AnalysisControllers}</td>
</tr>
<tr>
<td>Facades</td>
<td>{ModelingFacade, AnalysisFacade}</td>
</tr>
<tr>
<td>AnalysisControllers</td>
<td>{}</td>
</tr>
<tr>
<td>DataLayer</td>
<td>{FacadeExtension, GargoyleFacade, ControllerExtensions, GargoyleControllers, EntityFactory, GargoyleDomainAccessObjects, GargoylePersistentDomainObjects, Utils}</td>
</tr>
</tbody>
</table>

Table 6.1.: Architectural entities defined for the MosAIC web application

architecture description of subject software system MosAIC.

3. Creation of rules extracted from the architecture description given in the paper and depicted in the figures 6.3 and 6.2

4. Replay of the monitored communication and execution of the validation feature of the CIC application in order to retrieve all classes engaged in the execution of the user function validate.

5. Creation of code-level entities mapping the actual classes engaged and establishment of the include relations to their including architectural entities.

6. Replay of the monitored communication and execution of the validation feature of the CIC application in order to retrieve validation results.

7. Summary of the results and discussing them with the author of MosAIC to eliminate erroneous communication rules, create missing rules and fine tune the mappings.

8. Replay of the monitored communication and execution of the validation feature of the CIC application a second time with updated rules.

9. Discussion about the findings with the author of MosAIC.

The architectural information defined in the evaluation process consists of the architectural entities and their include relations, the code-level entities included by these architectural entities and the communication rules between the architectural entities. The architectural entities and their include relations are depicted in Table 6.1.

The list of code-level entities assigned to the classes engaged in the monitored execution to be validated consists of nearly 80 entities and is omitted for brevity. The
### 6. Evaluation

<table>
<thead>
<tr>
<th>Rule set</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>(source)</td>
<td></td>
</tr>
<tr>
<td>Presentation</td>
<td>* → Utils</td>
</tr>
</tbody>
</table>

#### Presentation

- `ModelingFacelets → ModelingPresenters`
- `AnalyzeisFacelet → AnalyzisPresenters`
- `AnalysisPresenters → Facades`
- `AnalysisPresenters → AnalysisControllers`
- `ModelingPresenters → ModellingControllers`
- `ModellingControllers → Facades`

#### PresentationToLogic

- `ModellingControllers → BackingBeans`
- `AnalysisPresenter → BackingBeans`
- `ModelingFacade → BackingBeans`
- `AnalysisFacade → BackingBeans`

#### Logic

- `AnalysisFacade → AnalysisControllers`

#### LogicToData

- `ModelingFacade → FacadeExtension`
- `AnalysisControllers → FacadeExtension`

#### Data

- `FacadeExtension → GargoyleFacade`
- `FacadeExtension → ControllerExtensions`
- `GargoyleFacade → GargoyleControllers`
- `GargoyleControllers → GargoyleDomainAccessObjects`
- `GargoyleControllers → EntityFactory`
- `GargoyleDomainAccessObjects → GargoylePersistentDomainObjects`
- `EntityFactory → GargoylePersistentDomainObjects`

**Table 6.2.: Rules defined for the MosAIC web application**

Rule sets marked with (*) were not retrieved from the architecture description but communicated by the author.

Rules defining the allowed communication of the target software system are depicted in Table 6.2.

The steps required to performed the validation of the MosAIC web application are described in more detail as follows:

1. First of all the Glassfish server hosting the MosAIC web application was prepared to monitor executions of the application. The preparation consisted of instructing the server to execute the application with the AspectJ Java agent and providing the required configurations for both AspectJ and the Kieker framework along with their libraries. Then a typical user function of the subject software system was executed and monitored.
2. The three top layers “Presentation layer”, “Logic layer” and “Data layer” of the MosAIC software system were retrieved from architecture description depicted in Figure 6.3. Figure 6.2 provides the five architectural entities included by the “Presentation layer”. The architectural entities included by the remaining two layers...
“Logic layer” and “Data layer” are again depicted in Figure 6.3. Additionally there is an implicit top level entity which functions as the border between “Presentation layer” and “Logic layer” which allows to exchange “Backing Beans” between these layers. This additional entity is not depicted as an entity in Figure 6.3 and was named “BackingBeans” reflecting its contents.

(3.) Architecture descriptions are the main sources to establish rules between different architectural entities. Analyzing the available descriptions five rule sets were created. The rule set “presentation” was extracted from Figure 6.2 and defines the allowed communication within the presentation layer. The rule sets “logic” and “data” defining allowed communication for the remaining two layer were extracted by Figure 6.3 which is also the source for the rule set “logicToData” describing the transition from the Data layer to the Logic layer. The rule set “presentationToLogic” defining the transition between Presentation and Logic layer which utilizes the BackingBeans entity was extracted from Figure 6.2 and Figure 6.3. Table 6.2 shows the rules defined for the MosAIC web application. However two rule sets named “utils” and “gargoyle” were only retrieved in the discussion with the author as recapitulated below.

(4.) After obtaining the monitored execution, creating architectural entities together with their include relations and setting up the communication rules between them the creation of code-level entities is the last task to complete the input for the desired validation. In order to reduce the workload of this task it was decided to not assign all of the approximate 300 classes of MosAIC to code-level entities. Instead only the classes involved in the monitored executing should be assigned. In order to retrieve all classes engaged in the execution the monitored communication was replayed and validated. Validating without having any code-level entities can only result in validations carrying a state of VALIDATION_FAILURE_* (with * being one of MISSING_CALLER, MISSING_CALLEE and NO_SOFTWARE_UNIT) as described in detail in Figure 5.1.4. The validation results indicating that the caller or callee is missing are of no interest at this point but the results of state VALIDATION_FAILURE_NO_SOFTWARE_UNIT are, as the collection of all callers and callees of these results reflect the code-level entities which are necessary for the validation.

(5.) The previous step revealed a list of methods engaged in the execution of the user function. Naturally none of them was assigned to a code-level entity as at this point no code-level entities were defined. In this evaluation an approach to map exactly one complete classes to code-level entities was followed. According to this approach the calling methods and the methods being called were aggregated by their classes. For each of these classes one distinct code-level entity was defined and the class was assigned to this entity. That yielded a definition of code-level entities with a 1-to-1 assignment of the classes to the entities. The exemplary extract of one of the code-level entity definitions shown in Source Code 6.1 shall illustrate this assignment.

```
1  <softwareUnit name='presenter.CategorizationBean'>
2       <filter>
3       <classFilter>presenter.CategorizationBean</classFilter>
```
6. Evaluation

The author of MosAIC was then asked to specify how these nearly 80 code-level entities were to be included by the different architectural entities listed before.

(6.) Completing the architectural information with the code-level entities as described before permitted a first validation attempt. The validation revealed that all callers and callees were assigned to code-level entities and all code-level entities were included by an architectural entity indicating that the architectural information sufficed. Furthermore much of the communication happened within single code-level entities or within one architectural entity which is always allowed as discussed in section 4.9. Other monitored communication followed nine of the defined communication rules. Also—as no denying rules were defined—no contradictory rules were identified. The target software system however also partly communicated in opposition to the defined rules which was reported as 17 missing rules.

<table>
<thead>
<tr>
<th>Calling entity</th>
<th>Called entity</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModelingPresenters</td>
<td>FacadeExtension</td>
<td>1</td>
</tr>
<tr>
<td>ModelingPresenters</td>
<td>AnalysisFacade</td>
<td>2</td>
</tr>
<tr>
<td>ModellingControllers</td>
<td>BackingBeans</td>
<td>22996</td>
</tr>
<tr>
<td>ModelingFacade</td>
<td>BackingBeans</td>
<td>562</td>
</tr>
<tr>
<td>ModelingPresenters</td>
<td>ModelingFacade</td>
<td>22</td>
</tr>
<tr>
<td>AnalysisFacade</td>
<td>BackingBeans</td>
<td>1</td>
</tr>
<tr>
<td>ModellingControllers</td>
<td>GargoylePersistentDomainObjects</td>
<td>28746</td>
</tr>
<tr>
<td>ModelingPresenters</td>
<td>AnalysisControllers</td>
<td>50</td>
</tr>
<tr>
<td>BackingBeans</td>
<td>GargoylePersistentDomainObjects</td>
<td>9007</td>
</tr>
<tr>
<td>ModelingPresenters</td>
<td>Utils</td>
<td>1</td>
</tr>
<tr>
<td>FacadeExtension</td>
<td>GargoylePersistentDomainObjects</td>
<td>5053</td>
</tr>
<tr>
<td>ModelingPresenters</td>
<td>BackingBeans</td>
<td>81</td>
</tr>
<tr>
<td>AnalysisControllers</td>
<td>GargoylePersistentDomainObjects</td>
<td>2671</td>
</tr>
<tr>
<td>BackingBeans</td>
<td>GargoyleFacade</td>
<td>2</td>
</tr>
<tr>
<td>AnalysisFacade</td>
<td>GargoylePersistentDomainObjects</td>
<td>2</td>
</tr>
<tr>
<td>ModelingPresenters</td>
<td>GargoylePersistentDomainObjects</td>
<td>130</td>
</tr>
<tr>
<td>ModelingFacade</td>
<td>GargoylePersistentDomainObjects</td>
<td>87962</td>
</tr>
</tbody>
</table>

Table 6.3.: Violations after the first validation attempt

(7.) The summary of the validation results described before was then discussed with the author of the target software system in order to improve the definition of the entities and rules. In this discussion one missing mapping and the absence of several rules was recognized. The missing mapping resulted from mapping all classes of one package to the architectural entity “ModelingPresenters” while some of the classes should have been mapped to the “AnalysisControllers” entity. The first missing rules were to define
the transition between “Presentation layer” and “Logic layer” (that is calls to the “BackingBeans” entity mentioned in the third step). The reason for the absence of these rules was that it was not entirely clear if they should allow communication from the whole “Presentation Layer” to the “Backing Beans” or only from the two entities at the border to of the layer. After clarifying that missing rules should only span “ModellingControllers” and “AnalysisPresenters” at the border of the “Presentation Layer” the missing rules were added as the rule set “presentationToLogic”. Furthermore the “Utils” entity should be callable with no restrictions by any other entity. The newly created rule set “utils” consists of one rule permitting this. Finally similar rules allowing calls to all Gargoyle* entities should be added.

(8.) Executing the validation the second time with updated rules yielded to:

<table>
<thead>
<tr>
<th>Calling entity</th>
<th>Called entity</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModelingPresenters</td>
<td>FacadeExtension</td>
<td>1</td>
</tr>
<tr>
<td>ModelingPresenters</td>
<td>AnalysisFacade</td>
<td>2</td>
</tr>
<tr>
<td>ModelingPresenters</td>
<td>ModelingFacade</td>
<td>22</td>
</tr>
</tbody>
</table>

Table 6.4.: Violations after the second validation attempt

(9.) The violations listed in [Table 6.4] were discussed again with the person responsible for the MosAIC development. This result was accepted as correct violation findings. The calls between the ModelingPresenters entity and the ModelingFacade entity were assessed to be of lower severity than the other three violations.

6.3. CIC

Apart from the MosAIC web application CIC (more specifically the InfromationProcessor tool) itself was analyzed in order to evaluate it. The detailed description of the different CIC packages in [section 5.1] includes the naming of the individual dependencies of the packages. From this dependencies a number of rules were created in a way similar to the approach taken during evaluation of the MosAIC web application. The defined rules are given in [Table 6.5]. A second set of rules was defined to allow the communication between the loosely coupled components of CIC which are connected by observers. This set of rules is shown in [Table 6.6]. Furthermore the rules from both sets are depicted in the informal notation of this thesis in [Figure 6.4].

While the validation of CIC worked out as expected and no architectural violations where found within CIC the evaluation revealed problems regarding the informal notation and the number of required rules as follows. The informal notation was slightly adjusted because of the number of rules: The arrows are now depicted winding instead of angled as before and now two-headed arrows are used instead of two arrows when communication is allowed in both directions. However despite the adjustments the sheer number of rules makes it very hard to read the depiction. Both tables and the depiction are given at the end of this section as they require whole pages.
6. Evaluation

The enumerations ValidationState and ValidationResult.Permission which give information about the state of the validation and its result (ALLOWED, DENIED) are used at different places of the software. While this is not considered a violation of the architecture it requires that both enumerations are globally accessible which requires the rules $\star \rightarrow validation.ValidationResult.Permission$ and $\star \rightarrow validation.ValidationState$. These rules cannot be depicted with the informal notation presented in this thesis.

The first observation apart from the informal notation is that CIC as it depends on runtime information is unable to recognize inheritance relationships. For example it does not suffice to allow instance of UnitCatalog—which manages both ArchitectureUnits (architectural entities) and SoftwareUnits (code-level entities)—to communicate with the abstract base class Unit. It also does not suffice to allow communication with the concrete classes ArchitectureUnit and SoftwareUnit. Instead, in order to validate the communication of the UnitCatalog without false positives, rules allowing communication to all three classes are required. This is caused by the runtime monitoring which reports for each method invocation the implementing class which can be the actual class of an instance or each of its superclasses. This was resolved by creating architectural entities which include the whole inheritance tree e.g. the entity Unit that includes the classes: Unit, ArchitectureUnit and SoftwareUnit.

The second observation was that by creating these entities within their packages did not automatically allow communication from the rest of the package and vice versa. This the intended behavior of the validation as allowing all communications within an entity is generally not desired. However it also introduces a high number of rules which only allow packages to communicate with entities within them.

6.4. Summary

CIC is in its initial phase providing abilities to model a subject software system and validate runtime execution against the modeled software architecture. The evaluation revealed only one shortcoming of CIC regarding its metamodel which allows modeling a subject software system. The metamodel allowed only definitions of communication rules between two existing architectural entities. However the rule $\star \rightarrow utils$ which allows all entities to call the utils-entity requires to match all calling entities and not a specific one. The metamodel implemented in the “architecture” package described in subsection 5.1.2 was extended to also allow definition of such rules\(^2\) and the validation strategies were updated accordingly. The remaining shortcomings were not related to the metamodel or validation implemented by CIC but mostly to the lack of tool support for both its input and output. Shortcomings of CIC in its current state are classified as follows according to whether they affect the required input or provided output of CIC.

\(^2\)For the sake of completeness the reversal ($A \rightarrow \star$) that is rules allowing one entity unconstrained communications were also implemented.
Input related shortcomings

Currently the model of the software system must be created as XML definitions by hand. While these definitions can at least be validated while defining them—using the XML schema definitions CIC provides—the process remains inconvenient as it requires basic skills in XML as well as knowledge of the semantics to define entities and communication rules. On account of this the entity and rules definition in this evaluation was, even though guided, performed by the author of the thesis. Having a graphical user interface (GUI) which enables visual definition of entities would enable software architects to directly model their systems. Although the absence of such a GUI was known before performing the evaluation confirmed the need for it. However it is assumed that CIC provides a sufficient basis to implement such an GUI on top of it. The informal notation used in this thesis gives an example for visualizing code-level/architectural entities together with communication rules. CIC is currently unable to provide the structure of the source code in order to ease the mapping between the code and the code-level entities. During the evaluation a 1-to-1 mapping between classes and code-level entities was targeted. This mapping was accomplished by first executing the validation without any defined code-level entities (4.) to retrieve all classes engaged and than manually³ defining the code-level entities mapping the classes (5.). Extending CIC in a way to allow extracting the source code’s structure and offering entities of it as candidates for code-level entities through the GUI would presumably increase the value of the GUI even more. On the other hand adding importers to CIC for already formally defined models especially models defined through ADLs would allow validation of software systems without modeling them exclusively for the use of CIC. This could theoretically eliminate the need for an exclusive GUI only for CIC. However the MosAIC’s web application is not defined through an ADL which is true for the most real-life software system.

Output related shortcomings

Currently CIC provides only output through its ValidationTool application which publishes its validation results JSON-encoded in a XMPP chat room. The idea behind this is to allow other applications to retrieve the results by participating in the chat room and decoding the JSON-data. This allows different applications to process the validation results at the same time while being only loose coupled to CIC. However CIC does not provide graphical output of the validation’s outcome. Different graphical outputs are conceivable ranging from highlighting violated or missing rules in a GUI as described for the input to augmenting call visualizations like sequence diagrams. The current output of a validation result contains the calling method/function and the method/function being called together with their code-level and architectural entities. It was mentioned that besides methods or functions the code line responsible for the violation would be a valuable information. Obtaining the code line would also allow to connect CIC to an IDE in order to highlight violations directly in the source code. However the code

³In order to ease this the definition of 80 code-level entities was done semiautomatically using the ValidationTool’s output and some scripting on a Linux-shell.
line must be recorded by the monitoring framework underlying CIC. Unfortunately the currently used Kieker framework does not provide this information. Furthermore CIC also incorporates only basic statistical functionality which merely consists of aggregating the results by their validation state. Lastly CIC does not contain any metric to process the validation result and by this is unable to assess the whole subject software system beyond simply counting its validation results. For example the evaluation of the MosAIC web application yielded three kinds of violations with a total of 25 violations in more than one hundred thousand checked communication attempts. CIC itself can not assess the discrepancy between the architecture and the implementation reflected by these violations. It only provides information about the violations and leaves the assessment of the subject software system to the people responsible for it.

It is not easy to depict more complex architectures as a whole including their rules in the presented, informal notation. It should be analyzed if these problems can be solved by using hierarchies of architectures instead of whole architectures.
Figure 6.4.: CIC architecture with communication rules
Table 6.5.: Rules defined for CIC.

<table>
<thead>
<tr>
<th>Rule set</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>enums (*)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>* \rightarrow \text{validation}.ValidationResult.Permission</td>
</tr>
<tr>
<td></td>
<td>* \rightarrow \text{validation}.ValidationState</td>
</tr>
<tr>
<td>rules</td>
<td>\text{rules}.Rule \rightarrow \text{rules}</td>
</tr>
<tr>
<td></td>
<td>\text{rules}.RuleCatalog \rightarrow \text{rules}.Rule</td>
</tr>
<tr>
<td></td>
<td>\text{rules}.RuleCatalog \rightarrow \text{rules}</td>
</tr>
<tr>
<td></td>
<td>\text{rules}.Rule \rightarrow \text{architecture}.Unit</td>
</tr>
<tr>
<td>architecture</td>
<td>\text{architecture}.Unit \rightarrow \text{architecture}</td>
</tr>
<tr>
<td></td>
<td>\text{architecture}.SoftwareUnit \rightarrow \text{architecture}.UnitCatalog</td>
</tr>
<tr>
<td></td>
<td>\text{architecture}.Enhancer \rightarrow \text{architecture}.UnitCatalog</td>
</tr>
<tr>
<td></td>
<td>\text{architecture}.Enhancer \rightarrow \text{monitoring}.ExecutionRecord</td>
</tr>
<tr>
<td>monitoring</td>
<td>\text{monitoring}.ExecutionRecord \rightarrow \text{monitoring}.ExecutionRecord</td>
</tr>
<tr>
<td></td>
<td>\text{monitoring}.ExecutionRecord \rightarrow \text{architecture}.SoftwareUnit</td>
</tr>
<tr>
<td></td>
<td>\text{monitoring}.ExecutionRecord \rightarrow \text{validation}.ValidationResult</td>
</tr>
<tr>
<td>xmpp</td>
<td>\text{xmpp} \rightarrow \text{xmpp}.XmppCommunicator</td>
</tr>
<tr>
<td></td>
<td>\text{xmpp}.XmppCommunicator \rightarrow \text{xmpp}</td>
</tr>
<tr>
<td></td>
<td>\text{xmpp}.XmppCommunicator \rightarrow \text{marshalling}</td>
</tr>
<tr>
<td>validation</td>
<td>\text{validation}.Validator \rightarrow \text{validation}</td>
</tr>
<tr>
<td></td>
<td>\text{validation} \rightarrow \text{validation}.Validator</td>
</tr>
<tr>
<td></td>
<td>\text{validation}.ValidationResult \rightarrow \text{rules}.Rule</td>
</tr>
<tr>
<td></td>
<td>\text{validation}.ValidationResult \rightarrow \text{rules}.Rule</td>
</tr>
<tr>
<td></td>
<td>\text{validation}.ValidationResult \rightarrow \text{rules}.Rule</td>
</tr>
<tr>
<td></td>
<td>\text{validation}.ValidationResult \rightarrow \text{rules}.Rule</td>
</tr>
<tr>
<td></td>
<td>\text{validation}.ValidationResult \rightarrow \text{rules}.Rule</td>
</tr>
<tr>
<td></td>
<td>\text{validation}.ValidationResult \rightarrow \text{rules}.Rule</td>
</tr>
<tr>
<td></td>
<td>\text{validation}.ValidationResult \rightarrow \text{rules}.Rule</td>
</tr>
<tr>
<td></td>
<td>\text{validation}.ValidationResult \rightarrow \text{rules}.Rule</td>
</tr>
<tr>
<td>importer</td>
<td>\text{importer} \rightarrow \text{importer}.ArchitectureImporter</td>
</tr>
<tr>
<td></td>
<td>\text{importer}.ArchitectureImporter \rightarrow \text{importer}</td>
</tr>
<tr>
<td></td>
<td>\text{importer}.RulesImporter \rightarrow \text{importer}</td>
</tr>
<tr>
<td></td>
<td>\text{importer}.RulesImporter \rightarrow \text{importer}</td>
</tr>
<tr>
<td></td>
<td>\text{importer}.ArchitectureImporter \rightarrow \text{architecture}</td>
</tr>
<tr>
<td></td>
<td>\text{importer}.ArchitectureImporter \rightarrow \text{architecture}</td>
</tr>
<tr>
<td></td>
<td>\text{importer}.ArchitectureImporter \rightarrow \text{architecture}</td>
</tr>
<tr>
<td></td>
<td>\text{importer}.RulesImporter \rightarrow \text{rules}</td>
</tr>
<tr>
<td>marshalling</td>
<td>\text{marshalling} \rightarrow \text{architecture}.Unit</td>
</tr>
<tr>
<td></td>
<td>\text{marshalling} \rightarrow \text{monitoring}.ExecutionRecord</td>
</tr>
<tr>
<td></td>
<td>\text{marshalling} \rightarrow \text{rules}.Rule</td>
</tr>
<tr>
<td></td>
<td>\text{marshalling} \rightarrow \text{validation}.ValidationResult</td>
</tr>
</tbody>
</table>
6.4. Summary

Rule set | Rule
--- | ---
connections | \( v.\text{Validator} \rightarrow x.\text{sendValidatedExecutionRecordsThroughXmpp} \)
 | \( x.\text{sendValidatedExecutionRecordsThroughXmpp} \rightarrow v.\text{ValidationResult} \)
 | \( x.\text{sendValidatedExecutionRecordsThroughXmpp} \rightarrow m.\text{ExecutionRecord} \)
 | \( m \rightarrow a.\text{enhanceNewExecutionRecord} \)
 | \( a.\text{enhanceNewExecutionRecord} \rightarrow a.\text{Enhancer} \)
 | \( a.\text{Enhancer} \rightarrow v.\text{validateEnhancedExecutionRecords} \)
 | \( v.\text{validateEnhancedExecutionRecords} \rightarrow v.\text{validator} \)

Table 6.6.: Rules defined for CIC (connections).

Abbreviations: \( m=\text{monitoring}, x=\text{xmpp}, v=\text{validation}, a=\text{architecture} \)
7. Conclusion

Research is formalized curiosity. It is poking and prying with a purpose.

Zora Neale Hurston

Contents

| 7.1. Future Work | 81 |

The presented concept allows for modeling architectures in form of generic architectural entities. These entities might be nested and can be part of communication rules either as calling entities or as entities being called. The architectural entities can be connected to elements of the source code through code-level entities. This allows to compare the communication within a software system during runtime with the defined communication rule. Doing so validates the software system with regard to the communication integrity property in form of a dynamical analysis. The presented concept is realized as a Java project named CommunicationIntegrityChecker (CIC). CIC consists of a class library which implements the entire functionality demanded by the concept. Two applications build on the class library allow to import architectures, read recorded runtime communications and validate the conformance of the communication. The realization was evaluated by validating another research project with it. The executed validation correctly found three different types of validation with an overall number of 25 violations by validating more than one hundred thousand method calls. As an additional evaluation the software itself was validated by itself. It is hoped that this approach will prove very useful for the detection of violations of the communication integrity thusly allowing for their removal as to close or at least minimize architectural gaps.

7.1. Future Work

CommunicationIntegrityChecker (CIC), the implementation of the concept presented in this thesis, is in its initial phase. CIC already provides the core functionality of the concept. However as each dynamic approach to analyze a software system, CIC suffers from the fact that it can only analyze the parts of a system which are active during the analysis. Integrating CIC into the unit tests of a software system would allow validation of larger portions of the system and to automate the validations. It seems that such an integration should be fairly easy to realize however this was not evaluated.
in this thesis. CIC currently supports only the Kieker framework as a provider of monitored executions. While Kieker provides abilities to monitor Java systems regardless of whether they are local or distributed it is limited to the Java technology. In order to enable CIC to monitor system based on other technologies CIC should be extended to support further monitoring frameworks. Apart from supporting only one monitoring framework CIC lacks tool support to offer a more convenient handling. A desirable tool support would cover both the input and output of CIC. For the input graphical creation and manipulation of architectural data and communication rules would allow comfortable modeling of architectures. Such tools may directly use the UnitCatalog and RuleCatalog of CIC. They however might also be decoupled from the CIC project and create architectural XML definitions, like the currently handmade architectural inputs, to be imported by CIC. For the output different objectives require different tools. For example in order to perform quality assurance by preventing architectural gaps to emerge the communication integrity validation should be regularly executed, optimally on every change of the architecture description or the realization. Furthermore any violation should be instantly and precisely reported to the person responsible. Ideally the reporting would be integrated into the same environment in which the change causing the violation was performed, that is the IDE of the programmer or the architecture modeling environment of the software architect.

On the other hand performing in-depth analysis of a software system would require tools able to present validation results by other means. For example it would be desirable to augment approved dynamic views of software systems like sequence diagrams or activity diagrams with architectural information and validation results. Lastly the presented concept and hence CIC do not incorporate a metric to assess identified violations. That prevents CIC to rate the quality of software system or to qualify the gap between conceptual and concrete though architecture although it is able to detect violations of the communication integrity.

Finally the validation ability of CIC is based on architectural (not code-level) entities and is hence decoupled from the monitored execution data. This allows to create other tools to produce input data to be validated by CIC. These input data might even originate from static analysis.

---

1 Such regular validations should probably be build upon the aforementioned unit test integration.
2 The LISA toolkit already provides such an integration Figure 3.3.
A. Appendix One

```xml
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.swc.rwth-aachen.de/CommunicationIntegrityChecker/architecture"
  xmlns:tns="http://www.swc.rwth-aachen.de/CommunicationIntegrityChecker/architecture"
  elementFormDefault='qualified'>
  <complexType name="units">
    <choice maxOccurs="unbounded" minOccurs="0">
      <element name="architectureUnit" type="tns:architectureUnit"></element>
      <element name="softwareUnit" type="tns:softwareUnit"></element>
    </choice>
  </complexType>
  <complexType name="softwareUnit">
    <sequence>
      <element name="filter" type="tns:filterChoice"></element>
    </sequence>
    <attribute name="name" type="ID"></attribute>
  </complexType>
  <complexType name="architecture">
    <sequence></sequence>
  </complexType>
  <complexType name="architectureUnit">
    <sequence>
      <element name="units" type="tns:units" maxOccurs="unbounded"
        minOccurs="0">
      </element>
    </sequence>
    <attribute name="name" type="ID"></attribute>
  </complexType>
  <element name="architecture" type="tns:units"></element>
  <element name="units" type="tns:units"></element>
  <complexType name="filterChoice">
    <choice>
      <element name="classFilter" type="tns:classFilter"></element>
      <element name="regexFilter" type="tns:regexFilter"></element>
      <element name="globFilter" type="tns:globFilter"></element>
    </choice>
  </complexType>
</schema>
```
A. Appendix One

```xml
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.swc.rwth-aachen.de/CommunicationIntegrityChecker/rules"
  elementFormDefault="qualified"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://www.swc.rwth-aachen.de/CommunicationIntegrityChecker/rules">
  <complexType name="rules">
    <sequence>
      <element name="allowUnruled" type="tns:allowUnruled"></element>
      <element name="allowUnenhanced" type="tns:allowUnenhanced"></element>
      <element name="ruleSet" type="tns:ruleSet"
        maxOccurs="unbounded" minOccurs="0"></element>
    </sequence>
  </complexType>
  <complexType name="ruleSet">
    <choice maxOccurs="unbounded" minOccurs="0">
      <element name="sameArchitectureUnitRule"
        type="tns:sameArchitectureUnitRule"></element>
      <element name="sameSoftwareUnitRule"
        type="tns:sameSoftwareUnitRule"></element>
      <element name="directedBiCommunicationRule"
        type="tns:directedBiCommunicationRule"></element>
      <element name="calleeCommunicationRule"></element>
    </choice>
  </complexType>
</schema>
```
<complexType name="ruleChoice">
  <choice maxOccurs="unbounded" minOccurs="0">
    <element name="sameSoftwareUnitRule" type="tns:sameSoftwareUnitRule"></element>
    <element name="sameArchitectureUnitRule" type="tns:sameArchitectureUnitRule"></element>
    <element name="directedBiCommunicationRule" type="tns:directedBiCommunicationRule"></element>
  </choice>
</complexType>

<complexType name="sameSoftwareUnitRule"></complexType>

<complexType name="sameArchitectureUnitRule"></complexType>

<complexType name="directedBiCommunicationRule">
  <sequence>
    <element name="caller" type="tns:namedItem"></element>
    <element name="callee" type="tns:namedItem"></element>
  </sequence>
  <attribute name="permission" type="tns:permission" use="optional" default="ALLOWED"></attribute>
  <attribute name="priority" type="int" use="optional" default="0"></attribute>
</complexType>

<complexType name="calleeCommunicationRule">
  <sequence>
    <element name="callee" type="tns:namedItem"></element>
  </sequence>
  <attribute name="permission" type="tns:permission" use="optional" default="ALLOWED"></attribute>
  <attribute name="priority" type="int" use="optional" default="0"></attribute>
</complexType>

<complexType name="callerCommunicationRule">
  <sequence>
    <element name="caller" type="tns:namedItem"></element>
  </sequence>
  <attribute name="permission" type="tns:permission" use="optional" default="ALLOWED"></attribute>
  <attribute name="priority" type="int" use="optional" default="0"></attribute>
</complexType>

<complexType name="namedItem">
  <attribute name="name" type="NCName"></attribute>
</complexType>
A. Appendix One

<complexType name="allowUnruled">
  <attribute name="value" type="boolean" use="required"></attribute>
</complexType>

<complexType name="allowUnenhanced">
  <attribute name="value" type="boolean" use="required"></attribute>
</complexType>

<simpleType name="permission">
  <restriction base="string">
    <enumeration value="ALLOWED"></enumeration>
    <enumeration value="DENIED"></enumeration>
  </restriction>
</simpleType>

Source Code A.2: rules.xsd
Bibliography


Bibliography


