Ontology-Based Architecture Enforcement:

Defining and Enforcing Software Architecture as a Concept Language using Ontologies and a Controlled Natural Language

An der Universität Hamburg eingereichte

Dissertation

zur Erlangung des akademischen Grades
Dr. rer. nat.

Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
der Universität Hamburg

vorgelegt von

Sandra Schröder

Hamburg, 2020

Datum der Disputation:	20. Oktober 2020
Gutachter	Prof. Dr.-Ing. Matthias Riebisch
	Prof. Dr. rer. nat. Horst Lichter

Abstract

The software architecture can be understood as a prescriptive abstraction of a software system. This means, it prescribes structural and behavioral *architecture rules* that need to be respected during the entire software development life cycle. During development and maintenance activities, there is a high risk of introducing *architecture violations* that are a result of disobeying the architecture rules. Violations that are not removed, accumulate over time. This results in a phenomenon called *architecture erosion* describing a gap between the *intended* architecture and the *implemented* architecture. Due to this gap, the software architecture can no longer be used as a valuable predictive and descriptive abstraction of a software system.

As an integral part of a software architect's work, *architecture enforcement* aims to ensure that the software architecture is implemented as intended, to avoid or at least minimize architecture erosion. Architecture enforcement has two goals, namely 1) ensuring the agreement on design decisions with stakeholders and 2) checking the conformance between the implementation and the software architecture.

To achieve the first goal, the software architect needs to ensure that developers share the same understanding about the software architecture. The software architect thereby needs to manage possibly conflicting views in the team that may impede the consensual agreement on the software architecture. For this, he - together with the developers - needs to create a *language* used to consistently talk about the software architecture.

To achieve the second goal, the software architect validates the rules against the implementation. With the increasing size of the software system, it is often not feasible to manually check whether rules are violated by the implementation. That is why, the software architect makes use of powerful tool-support. Those tools provide a formal language, so that architecture rules can be automatically verified. However, those formal languages are still neglected by practitioners, since they are perceived as unusable. Additionally, formal languages are often not flexible enough so that not all architectural concerns can be appropriately expressed and validated. As a consequence, crucial violations against architecture rules cannot be detected.

In this dissertation an approach is developed that aims for *supporting software architects and developers in the architecture enforcement process.* To address the first goal of architecture enforcement, the approach provides a means for explicitly capturing a common language about the software architecture using *ontologies* and *description logics.* This language is called the *architecture concept language.* By setting up architecture rules in this language, the main architectural abstractions and thereby the software architecture are defined. Since these formalisms are not restricted to a specific application domain, they flexibly represent any architectural abstractions needed to describe the software architecture. As an understandable natural-language frontend, the *Architecture Controlled Natural Language (ArchCNL)* is proposed, a *Controlled Natural Language*, that is used to define the architecture concept language. Architecture rules are defined as natural-language sentences that are automatically verified against the implementation using an ontology-based conformance checking approach *ArchCNLCheck* that is also proposed in this thesis.

An evaluation with three industrial projects shows that the approach is able to formalize a great variety of rules found in projects due to its flexibility to represent architectural abstractions. Additionally, a focus group has been conducted where the perceived applicability of the approach has been evaluated. The study shows that practitioners perceive the approach as understandable and usable. Furthermore, an evaluation with two open-source systems shows that the approach can reliably detect violations of architecture rules in the implementation.

Kurzzusammenfassung

Softwarearchitektur kann als eine vorschreibende Abstraktion eines Softwaresystems verstanden werden. Dabei definiert die Softwarearchitektur sogenannte *Architekturregeln*, die die Struktur und das Verhalten des Softwaresystems einschränken. Diese Regeln müssen im gesamten Lebenszyklus der Software eingehalten werden. Der Lebenszyklus der Software wird durch ständige Änderungen bestimmt. Dies erhöht das Risiko, dass vorgegebene Architekturregeln verletzt werden. Architekturverletzungen, die nicht behoben werden, sammeln sich im Laufe der Zeit an. Dies resultiert in sogenannter *Architekturerosion*, welche die Lücke zwischen der geplanten und der tatsächlich implementierten Softwarearchitektur beschreibt. Eine erodierende Softwarearchitektur verliert ihren Wert als vorhersagende und beschreibende Abstraktion eines Softwaresystems.

Eine wichtige Verantwortung des Softwarearchitekten ist es sicherzustellen, dass die Softwarearchitektur wie geplant in der Implementation umgesetzt wurde. Dabei verfolgt der Softwarearchitekt zwei Ziele: 1) Er stellt sicher, dass alle Beteiligten der Softwarearchitektur zustimmen und 2) er überprüft, ob die Architekturregeln in der Implementation eingehalten wurden.

Um das erste Ziel zu erreichen, muss der Softwarearchitekt gewährleisten, dass alle Beteiligten das gleiche Verständnis von der Softwarearchitektur haben. Dabei muss er unterschiedliche, möglicherweise zueinander konkurrierende Meinungen und Ansichten der Entwickler zusammenbringen und eine Basis für ein gemeinsames Verständnis schaffen. Dafür muss der Softwarearchitekt zusammen mit den Entwicklern eine gemeinsame Sprache entwickeln, die genutzt wird, um die Softwarearchitektur zu beschreiben.

Zur Erreichung des ersten Ziels nutzt der Softwarearchitekt Werkzeuge, um die Übereinstimmung der Implementierung mit den definierten Architekturregeln (Konformanz) automatisiert zu prüfen. Eine manuelle Überprüfung der Konformanz ist oft aufgrund der Größe des Softwaresystems kaum möglich. Für diese Aufgabe existieren einige Werkzeuge, die den Softwarearchitekten unterstützen können. Diese Werkzeuge bieten formale Sprachen, um die Architekturregeln zu formalisieren und anschließend automatisiert zu prüfen. Oft sind diese formalen Sprachen zu umständlich zu benutzen und die resultierende Regelformalisierung ist nicht selten unverständlich. Die Formalisierung kann häufig nur von einem Experten, jedoch nicht von allem Teammitgliedern benutzt und verstanden werden. Aufgrund dessen finden formale Sprachen nur selten ihren Einsatz in der Praxis.

Diese Dissertation stellt einen Ansatz vor, der Softwarearchitekten und Entwickler bei der Einhaltung der Softwarearchitektur helfen soll. Der Ansatz ermöglicht Softwarearchitekten und Entwicklern eine gemeinsame Sprache über die Softwarearchitektur explizit zu definieren und im gesamten Lebenszyklus der Software zu nutzen. Diese Sprache wird in dieser Dissertation *Architekturkonzeptsprache* genannt. Sie definiert die wichtigsten Architekturelemente, die zur Beschreibung der Softwarearchitektur benötigt werden. Sie definiert zusätzlich Regeln für die Architekturelemente, die in der Implementierung eingehalten werden müssen. Der Ansatz basiert auf *Ontologien* und *Beschreibungslogik.* Diese Formalismen sind nicht an eine bestimmte Domäne gebunden und können somit genutzt werden, um Architekturelemente flexibel und formal zu beschreiben. Zusätzlich wird in der Dissertation eine kontrolliert natürliche Sprache (engl.: *Controlled Natural Language*) entwickelt, die der Softwarearchitekt und die Entwickler nutzen, um die Architekturkonzeptsprache zu definieren. Sie dient als eine verständliche und benutzbare Schnittstelle zu den Formalismen. Der Softwarearchitekt beschreibt dabei Architekturregeln als natürlichsprachliche Sätze, die die Architekturkonzeptsprache repräsentieren und automatisiert gegen die Implementation geprüft werden können.

Die Evaluationen zeigen, dass der Ansatz Softwarearchitekten und Softwareentwickler effektiv beim Einhalten der Softwarearchitektur unterstützen kann. Der Ansatz wurde mit drei Industrieprojekten evaluiert. Zu jedem Projekt wurden Architekturregeln gesammelt. Diese wurden mit dem Ansatz formalisiert. Es hat sich gezeigt, dass der Ansatz einen großen Teil der Architekturregeln formalisieren kann. Regeln, die mit existierenden Ansätzen nicht ausgedrückbar sind, können mit dem hier vorgstellten Ansatz formalisiert werden. In einer Fokusgruppe mit 12 erfahrenen Softwareentwicklern wurde die wahrgenommene Anwendbarkeit des Ansatzes evaluiert. Qualitative und quantitative Analysen zeigen, dass der Ansatz als verständlich und benutzbar wahrgenommen wird. Anhand zweier Open-Source-Systeme wird gezeigt, dass der Ansatz zuverlässig relevante Architekturverletzungen aufdecken kann.

Publications

Conference Publications

- **[Sch16]** Sandra Schröder, Mohamed Soliman, Matthias Riebisch: *Architecture Enforcement Concerns and Activities - An Expert Study.* Proceedings of the 10th European Conference on Software Architecture: 247-262 (2016)
- **[Ger16]** Sebastian Gerdes, Stefanie Jasser, Matthias Riebisch, Sandra Schröder, Mohamed Soliman, Tilmann Stehle: *Towards the Essentials of Architecture Documentation for Avoiding Architecture Erosion.* Proccedings of the 10th European Conference on Software Architecture Workshops: 8 (2016)
- **[Sch17]** Sandra Schröder, Matthias Riebisch: *Architecture Conformance Checking with Description Logics.* 11th European Conference on Software Architecture, Companion Proceedings: 166-172 (2017)
- **[Rac18]** Paula Rachow, Sandra Schröder, Matthias Riebisch: *Missing Clean Code Acceptance and Support in Practice - An Empirical Study.* 25th Australasian Software Engineering Conference: 131-140 (2018)
- **[Sch18a]** Sandra Schröder, Matthias Riebisch: *An Ontology-based Approach for Documenting and Validating Architecture Rules.* Proceedings of the 12th European Conference on Software Architecture, Companion Proceedings: 52:1-52:7 (2018)
- **[Sch19a]** Sandra Schröder, Georg Buchgeher: *Applicability of Controlled Natural Languages for Architecture Analysis and Documentation: An Industrial Case Study.* Proceedings of the 13th European Conference on Software Architecture, Companion Proceedings: 190-196 (2019)
- **[Sch19b]** Sandra Schröder, Georg Buchgeher: *Formalizing Architectural Rules with Ontologies - An Industrial Evaluation.* 26th Asia-Pacific Software Engineering Conference (2019)

Journal Publications

- **[Sch18b]** Sandra Schröder, Mohamed Soliman, Matthias Riebisch: *Architecture Enforcement Concerns and Activities - An Expert Study.* Journal of Systems and Software 145: 79-97 (2018)

Poster Publications

- **[Sch19c]** Sandra Schröder, Georg Buchgeher: *Discovering Architectural Rules in Practice.* Proceedings of the 13th European Conference on Software Architecture, Companion Proceedings: 10-13 (2019)

Danksagung

Für die Unterstützung bei der Anfertigung meiner Dissertation möchte ich den folgenden Personen danken:

- Prof. Dr. Matthias Riebisch für die Übernahme des Erstgutachtens und für die Möglichkeit, meine Dissertation in der Gruppe SWK durchführen zu können.
- Prof. Dr. Horst Lichter für die Bereitschaft, das Zweitgutachten zu übernehmen.
- Dr. Georg Buchgeher für die hervorragende Zusammenarbeit sowie für die Möglichkeit, die Evaluation meines Ansatzes am Software Competence Center Hagenberg durchführen zu können.
- Den Mitarbeitern des Software Competence Center Hagenberg für die Beteiligung an der Fokusgruppe und an den Interviews zur Evaluierung meines Ansatzes.
- Den Interviewteilnehmern für die Beteiligung an der Studie zum Thema *Architecture Enforcement in Practice*.
- Den Mitarbeitern des DFG-Projekts *Performance Conscious HPC (PeCoH)* für die hervorragende Zusammenarbeit.
- Allen aktuellen und ehemaligen Mitgliedern der Gruppe SWK, insbesondere meinen KollegInnen Leif Bonorden, Sebastian Gerdes, Stefanie Jasser, Paula Rachow, Dr. Mohamed Soliman, Dr. Tilmann Stehle und Yibo Wang.
- Dr. Christian Frédéric Adolff, Stefanie Jasser, Paula Rachow und Marion Wiese für die Unterstützung bei der Durchsicht dieser Arbeit.
- Meinem Lebensgefährten Dr. Christian Frédéric Adolff für die wertvollen, morgendlichen Diskussionen über Softwarearchitektur auf dem Weg zur Arbeit.
- Meinen Eltern Holger Schröder und Johanna Hennig-Schröder für den Rückhalt und die Unterstützung während meines gesamten Studiums.

Zuletzt wünsche ich den aktuellen und zukünftigen Doktoranden viel Erfolg im Verlaufe ihrer Promotion.

Contents

List of Figures

List of Tables

List of Acronyms

If you stick a Babel fish in your ear you can instantly understand anything said to you in any form of language.

(Adams, Douglas, 1952-2001. (1980). The Hitchhiker's Guide to the Galaxy. New York: Harmony Books.)

1. Introduction

1.1. Context and Problem Statement

Humans communicate with each other by using languages. The natural language, either spoken or written, constitutes the main means of communication between humans to exchange knowledge or to express needs, opinions, emotions, and beliefs.

Software development is a process greatly determined by communication between humans [AJLN08, Wei71]. It has been shown that communication problems can severely affect the success of a software project [ARE96, TKLVV15]. In order to specify and build software systems, many people with different roles and perspectives are required to communicate with each other [PSV94]. For example, studies by DeMarco et al. show that developers in large projects typically spend about 70% of their time communicating and working with other stakeholders [DL13]. Stakeholders include – amongst others – customers, domain experts, project managers, developers, testers, and software architects [Kru00]. All these stakeholders need to share a common understanding about the domain problems and how software aims to solve them [RR00]. For this, languages are a key element. Languages with different properties, e.g., formality or notations, are used for communication and specification [Stö17]. For example, stakeholders use natural language for face-to-face communications or for written communications, e.g., via E-Mail, chats, issue descriptions, or via exchanging documents [HSBA10] [KDV07]. *Modeling languages* [CFJ$^+$16], such as the Unified Modeling Language [RJB04], can be used to graphically specify the problem space a software system should be developed for. The diagrams are also used to explain the software system to stakeholders [Stö17].

Software development not only requires communication between humans; it also requires communication between humans and computers. Developers apply programming languages in order to transform – informally specified – requirements into an implementation, i.e., source code that a computer can understand and execute.

Software architecture is situated between software specifications and implementation. It has long been acknowledged as a key means for the communication between stakeholders during the software development life cycle [BCK12]. A system's software architecture describes its architectural design decisions. These decisions have the highest impact on the system's quality and are hard to change after their implementation [BCK12]. Architectural design decisions are usually taken by an experienced software engineer, who might play the *software architect* role in a project [McB07]. For example, the Rational Unified Process (RUP) [Kru00] specifies a dedicated role for a software architect. In agile processes (e.g., Scrum [SB01]), architects do not have a dedicated role, and could be involved in doing other tasks as well [Bab09, Fow03, Kru08]. *Architecture enforcement* is one of those challenging architecture duties being concerned with the correct and seamless implementation of architectural design decisions [Zim09]. Architecture enforcement requires software architects to thoroughly share and communicate the architectural design decisions and resulting architecture design with the developers who eventually implement the decisions. This is necessary, because the software architecture imposes several constraints on subsequent refinement phases, such as design and implementation. As Fairbank states, *"architecture constrains programs"* [FG10]. These constraints should act as *guide rails* and "*are essential in the construction of a system, in its ability to perform its job, and in the ability to maintain it over time*" [FG10]. In this thesis, these constraints are called *architecture rules.* Due to several factors such as cost and time pressure, developers might introduce sub-optimal solutions that violate such architecture rules [DSB12]. If violations against architecture rules remain unrepaired and accumulate, it cannot be guaranteed anymore that the software system is still able to fulfill the requirements. The implementation drifts away from the software architecture and it looses its value as a predictive and descriptive abstraction of the software system [KN16]. That is why, the software architect has to make sure that these architecture rules are respected by developers by enforcing the architectural design decisions and related architecture rules.

In order to enforce the software architecture, the software architect strives to ensure the consensus of developers on architectural design decisions. For this, the architect may make use of architecture documentation [CGB+10] to provide guidance during enforcement and to maintain a prescriptive description of the software architecture, e.g., in form of architecture rules. Ideally, an architecture should be well documented, "*with at least one static and one dynamic view using an agreed-on notation that all stakeholders can understand with a minimum of effort*" [BCK12]. An essential activity in architecture enforcement is to compare the architecture documentation with the implementation in order to make sure that they still align with each other or to reveal deviations between the two. This process is called *architecture conformance checking* [KN16]. However, with the growing size of software systems, it becomes time consuming to manually analyze the implementation for the adherence to architectural design decisions. That is why, architecture conformance checking should be tool-supported. This requires the architecture documentation to be described formally so that the description can be interpreted and processed by a machine. A lot of formal approaches have been developed to describe software architecture, such as Architecture Description Languages [MT00] or Alloy [Jac12], and to perform architecture conformance checking. However, two challenges arise when using these approaches:

Inflexible and Restrictive Languages: Software architects and developers implicitly decide for a language they describe the software architecture of the software system with [Vö10] [Smo02]. The language defines a vocabulary containing terms of the main architecture abstractions that have a specific meaning. By doing this, software architects and developers establish a shared *ontology.* Generally, an ontology is an *"explicit specification of a shared conceptualization"* [Gru95] and defines a vocabulary that describes concepts and relations that are representational for a domain. In the context of software architecture, it captures the consensual agreement on architecture abstractions that characterize their software architecture. The ontology explicitly conceptualizes the mental model of the team members about the software architecture. Such an ontology is project-specific. This means, software architects and developers use a terminology and define a meaning of the terminology that is specific to the software architecture defined in that project. However, as mentioned by Woods et al,. most existing modeling languages are restrictive with respect to the provided vocabulary and impose a particular architecture model on the architect [WH05] that is often not appropriate. Existing tools do not allow for customizing the language with a user-defined vocabulary. As a result, it is either not possible to describe all crucial architectural aspects or one is forced to use the provided language instead of the one inherent to the project. In the best case, this only builds a new but consistent language where the vocabulary deviates from the language used in the project. In the worst case, the original intention of the actual architecture description gets lost during the translation to the forced language.

Incomprehensible formalization: As mentioned in [ABO+17] and [MLM+13] there is still a lack of approaches that provide usable and readable architecture formalizations. Moreover, formal approaches to describe software architecture typically do not integrate well in the toolchain used by the developers. Therefore, architecture rules cannot be appropriately preserved and documented. Humans may prefer natural language descriptions enriched with informal drawings that sketch the main idea of the architecture. While this form of documentation is simple, widely understandable for humans, and expressive, natural language descriptions do not provide the unambiguity of formal descriptions. Additionally, it is not possible to use these descriptions for tool-supported analyses. In case that architectures are documented, typically only natural language descriptions are used or incomplete formal descriptions are supported with natural language explanations. However, the formalization and the informal description are prone to deviate from each other. With the recent development of Natural Language Processing (NLP) [JM00], attempts are made to make computers understand natural-language descriptions. However, NLP is far from being a solved problem and methods are still unreliable in making computers fully understand natural language [Kuh10].

1.2. Goals of the Thesis

Architecture enforcement is a complicated process that is of practical importance and has increasingly attracted attention in research [Zim09, CLN15, TV09, Nic18, KZ10, MCH16]. To appropriately address the challenges of architecture enforcement, methods and tools are required to effectively support a software architect to achieve the goals of architecture enforcement. This thesis aims to develop an approach that supports software architects in the architecture enforcement process. Based on the challenges described in the previous section, the approach is based on the following hypothesis:

Explicitly establishing and capturing a consistent language, i.e., an ontology, about the software architecture supports software architects and developers in the architecture enforcement process.

Ideally, the novel approach shall support and enhance the current practices architects perform during architecture enforcement. For this, it is necessary to characterize the architecture enforcement process. This means, it needs to be investigated which aspects software architects consider as most important and which activities they perform in order to enforce these aspects. That is why, the first goal is stated as follows:

Goal: G1

Identifying the major concerns and activities of a software architect in the architecture enforcement process.

Secondly, this thesis aims for developing an approach that appropriately addresses the limitations of current approaches with respect to their flexibility, so that crucial aspects considered by architects can be formalized. That is why, the next goal of the thesis is defined:

Goal: G2

Developing an approach that allows for a formal, flexible, and understandable definition of the project-specific language used by software architects and developers in order to support establishing a common understanding of the software architecture. The approach helps to define a formal meaning of the terms used in the context of the project.

Once formalized, the approach shall support software architects to automatically validate whether the established language is consistently used by the developers. That is why, the third goal of the thesis is defined:

Goal: G3

Developing an approach that allows for the tool-supported validation of the project-specific language against the source code of the software system. The approach shall help to find indications in the source code where the language has been violated, so that the software architect can take corresponding actions to address the violations appropriately.

In the following section, the solution proposal will be portrayed and the individual contributions according to the thesis goals are described.

1.3. Solution Proposal and Contributions of the Thesis

The approach that will be pursued in this thesis is to adopt *ontologies* [Gru95] for a formal and flexible definition of languages that software architects use to describe and to communicate the architecture. In the following, the individual contributions of this dissertation are summarized.

1.3.1. An Empirical Analysis of Architecture Enforcement in Practice

In order to address goal **G1**, an empirical study has been performed. This study encompasses interviews with 17 experienced software architects from industry working in different domains, e.g., the automotive, enterprise, and embedded domain. Using techniques from grounded theory [Gla78] and qualitative analysis [SC$^+$90, MB09], the industrial practice in the context of architecture enforcement is characterized. The study reveals architecture *enforcement concerns* and *enforcement activities*. The study and its results have been contributed to the *European Conference on Software Architecture* **[Sch16]** and an extension of the study has been published in the *Journal of Systems and Software* **[Sch18b]**.

1.3.2. Ontology-Based Definition of Architecture Concept Languages

This contribution addresses goal **G2**. The project-specific language that architects and developers use to describe their architecture is called *architecture concept language.* In this thesis, an ontology-based approach is proposed that allows for a formal and flexible definition of this language. Ontologies are a natural fit for representing and formalizing architecture concept languages. Ontologies are not restricted to specific concepts and relations. That is why, the software architect can flexibly define his language as needed in the project context. Furthermore, having an explicit representation of architecture concepts and relations supports the creation and preservation of a common language about the architecture [Vö10] that can be shared in the entire team. This idea is similar to the Domain Driven Design (DDD) approach [Eva04], where software developers and domain experts develop a so-called *ubiquitous language* which is also represented in the source code.

Unfortunately, ontologies and description logic require experts to design and maintain the language. Additionally, this logic-based representation does not integrate well into tool chains as they are used by developers [KAZ18]. That is why, the comprehensible natural-language frontend *Architecture Controlled Natural Language (ArchCNL)* is proposed in this thesis. It aims for facilitating the creation and formalization of architecture rules. *ArchCNL* is considered a so-called *Controlled Natural Language* (CNL). A CNL "*is a constructed language that is based on a certain natural language, being more restrictive concerning lexicon, syntax, or semantics, while preserving most of its natural properties*" [Kuh14]. CNLs integrate well with the description logic formalism and ontologies, so that they are frequently applied as knowledge representation languages or for ontology authoring, as in [Sch10]. Similarly, *ArchCNL* is restricted in its grammar and is grounded on ontology formalisms that have a well-defined syntax and semantics. This means, sentences written in *ArchCNL* can also be understood and processed by computers.

ArchCNL provides an executable specification language that is readable and understandable. The architecture concept language is specified by writing sentences with *ArchCNL.* Sentences define architecture rules that correspond to constraints of the architecture concept language on architecture concepts and relations. With *ArchCNL*, architecture rules can be read as natural language sentences that are widely comprehensible with an unambiguous meaning and that are also verifiable at the same time. Therefore, the approach facilitates an *understandable and verifiable architecture documentation.*

The proposed *ArchCNL* has been published in the *Workshop on Software Architecture Erosion and Architectural Consistency* **[Sch18a]**.

1.3.3. Ontology-Based Architecture Conformance Checking

The approach exploits efficient reasoning algorithms of ontologies in order to allow software architects to verify whether the language is consistently used in the source code. The approach proposes *ArchCNLCheck*, a process and tool that implements *ontology-based architecture conformance checking.* This contribution addresses goal **G3**. The approach uses the architecture rules defined by the architecture concept language and validates them against an ontology-based representation of the source code. For this, the approach provides ontologies that define concepts and relations that characterize source code artifacts. The source code is automatically transformed to this ontology-based representation.

The approach and an evaluation of its violation detection quality have been published in the *Workshop on Software Architecture Erosion and Architectural Consistency* **[Sch17]**, **[Sch18a]**.

1.3.4. An Empirical Evaluation of the Approach

The approach is applied on three industrial projects in order to evaluate its actual flexibility to formalize and represent architecture concept languages and corresponding architecture rules. The results show that the approach is able to formalize a great amount of complex architecture rules found in these industrial projects. Additionally, the perceived applicability of *ArchCNL* has been assessed in a focus group with 12 experienced software developers and software architects. The results of the study reveal that the approach provides a readable and understandable formalization of architecture rules and that the approach supports to establish and maintain a consistent vocabulary about the software architecture.

The ontology-based architecture conformance checking approach has been evaluated using two open-source systems. The goal of this evaluation is to assess its ability to detect relevant architecture violations by comparing the detection results with a ground truth of architecture violations of each software system at hand. The results of the evaluation show that the approach is able to find relevant violations. The approach even finds relevant violations that are not contained in the ground truth as confirmed by manual analysis.

The study results have been published in **[Sch19a]**, **[Sch19b]**, and **[Sch19c]**.

1.4. Thesis Outline

Figure 1.1 presents the contributions of the thesis, how they are organized into the respective chapters, and how they are related with each other. As can be seen, the thesis is organized into three main parts: an analysis of the state-of-the-practice of architecture enforcement, a description of the approach developed in this thesis, and an evaluation of the presented approach. Additionally, Figure 1.1 shows the respective research methods for the empirical study and for the evaluation conducted to evaluate the stated goals.

- Chapter 2 provides the fundamentals of the thesis. It first introduces basic terms and concepts of the software architecture field by providing fundamental definitions. Secondly, the theoretical foundations of description logic, ontologies, and CNLs are given. These foundations are necessary to develop the proposed approach.

- In Chapter 3, an empirical study on architecture enforcement in practice is presented. The empirical study is conducted by interviewing experienced software architects from practice. Using grounded theory, architecture *enforcement concerns* and *enforcement activities* are revealed. These findings characterize the enforcement process and additionally provide a motivation and the guiding principles for the approach developed in this thesis.

- Chapter 4 presents existing approaches for architecture conformance checking. Based on the findings from Chapter 3, criteria for the evaluation of conformance checking approaches are defined. These criteria describe necessary characteristics an approach must fulfill to appropriately support the architecture enforcement process. The limitations of existing approaches are used as a motivation for the need of a novel approach.

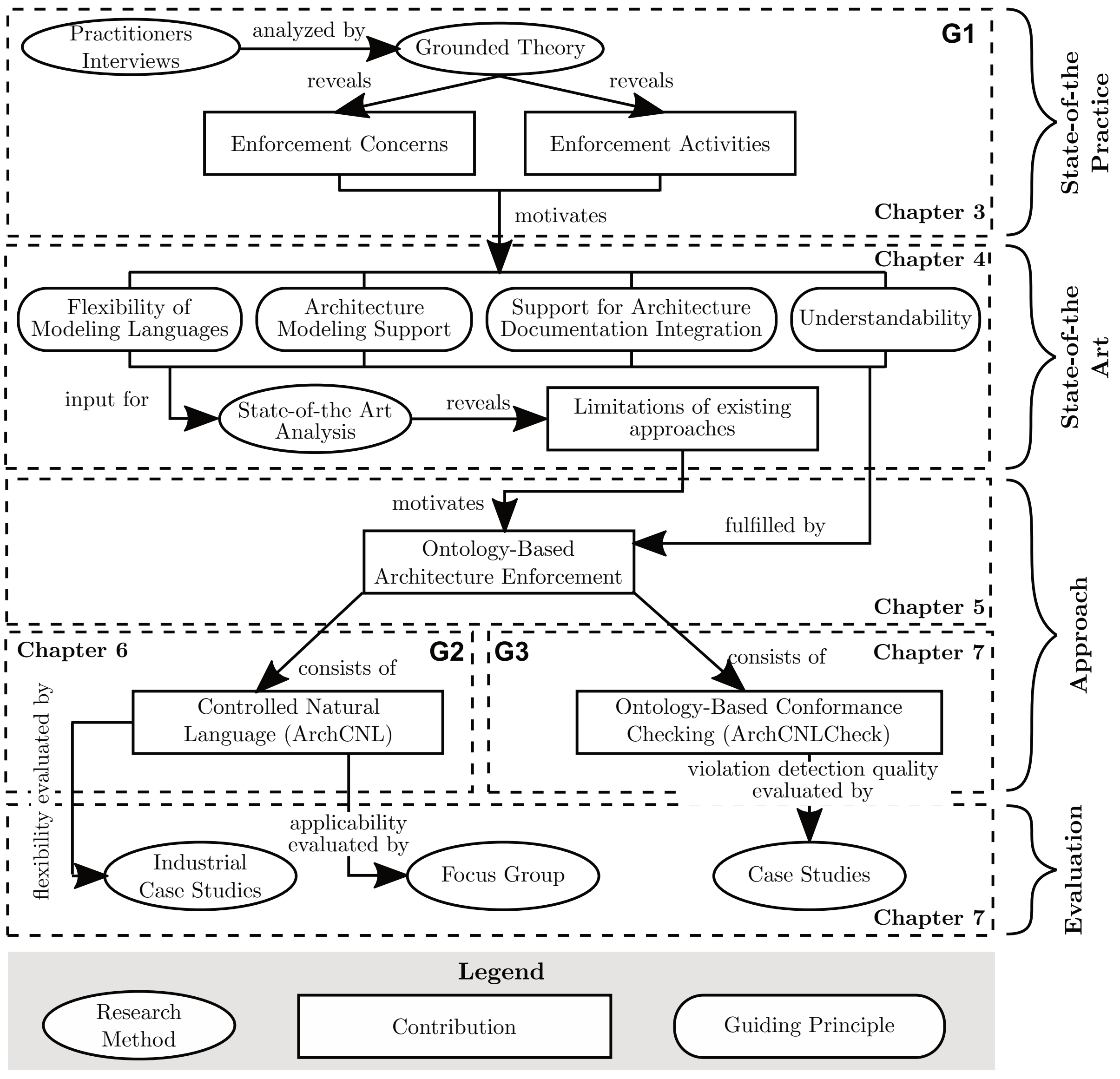


Figure 1.1.: *Contributions of the chapters as they are structured in this thesis and how they are related with each other.*

- Motivated and guided by the results revealed in Chapter 3 and in Chapter 4, Chapter 5 presents an overview of the developed approach that is called *ontology-based architecture enforcement*. It is shown how the approach fulfills the derived criteria from Chapter 4.

- Chapter 6 presents *ArchCNL* that builds the natural-language frontend to specify architecture concept languages. It is shown how sentences written in *ArchCNL* are mapped to architectural rule types and how a so-called *architecture ontology* representing the language is automatically generated from sentences written in *ArchCNL*.

- In Chapter 7, the ontology-based architecture conformance checking process is presented. This chapter includes several contributions. First, it presents three ontologies for modeling heterogeneous types of source code artifacts. Then, it is shown how the implemented architecture can be extracted from these ontology-based code representations by applying a rule-based approach based on the Semantic Web Rule Language (SWRL) [swr04]. Next, it is described how architecture violations are computed using description logic reasoners, how the results are again stored as an ontology. For this, an ontology representing a conformance check is designed and presented. Finally, the components of the toolchain implementing the ontology-based conformance checking process are described.

- The evaluation of the approach is presented in Chapter 8. The evaluation has the following goals: 1) evaluating the flexibility of the ontology-based architecture rule formalization using industrial case studies, 2) evaluating the applicability of the approach based on a focus group, and 3) evaluating the architecture violation detection quality of the ontology-based conformance checking process in order to verify whether its detection quality can compete with the quality of existing approaches. Additionally, a critical discussion on the approach is given. For this, the discussion again refers to the findings from the empirical study described in Chapter 3 and discusses how the approach actually supports the architecture enforcement process. Moreover, a critical discussion on the limitations of the approach is given.
- Chapter 9 concludes the thesis and summarizes the contributions. Additionally, it provides an overview on possible future work.

1.5. Note on Style

For the sake of simplicity and easier reading, only the masculine form has been used for the individual categories of people. However, it is assumed that this refers to all genders on equal terms.

2. Thesis Fundamentals

In this chapter, the fundamentals of the thesis are presented. First, the general process of architecture-centric development is described. This is necessary in order to understand the main activities conducted and artifacts produced in the architecture-centric development. Additionally, it is described how architecture enforcement is integrated in these activities. In this thesis, a formal means for describing software architecture is presented. That is why, existing modeling languages with different notations and levels of formality for software architecture description are presented in this chapter. Since the approach in this thesis implements architecture conformance checking, the main steps and terms of this process are described. The approach proposed in this thesis is based on the description logic formalism and ontologies. Therefore, the background on the description logic formalism, ontologies, Semantic Web technologies, and Controlled Natural Languages (CNLs) are described in the last part of this chapter.

2.1. General Process of Architecture-Centric Development

Every software system has an architecture regardless of whether it has been explicitly designed or it has emerged spontaneously from design decisions that have been made unconsciously. During software architecture design [HNS00, BCK12, Kru04b], software architects or developers make principal design decisions about a software system. Software architecture aims for making design decisions explicit. Consequently, this enables prediction, analysis, and governance [KN16]. The need for making such decisions is driven by concerns expressed by stakeholders. These concerns are properties stakeholders expect a software system to have. As defined in the ISO/IEC 42010 2011 standard, those concerns are, besides others, *functional* and *non-functional requirements* (or quality requirements). Functional requirements define the features of a software system, i.e., what a software system should provide, so that users can accomplish their tasks. For example, a software system for an e-commerce marketplace must provide the functionality to store and delete account information of customers. Non-functional requirements define the quality of the software system. For example, performance, security, or modifiability are non-functional requirements. Software architecture is the basis for every software system [CKK$^+$03] and constitutes the bridge between system requirements and the implementation of the software system [Gar03]. Software architecture has a great impact on how well a software system fulfills its non-functional requirements. Making adequate architectural design decision is crucial, since they enable or inhibit quality attributes of the system. This means, choices about how software architecture is designed greatly influence to which extent quality attributes are achieved by the software system. Choosing an appropriate software architecture can help to achieve the desired quality attributes, while making wrong decisions about the software architecture implies a high risk that the software system does not fulfill these requirements [Gar03].

2.1.1. Definitions of Software Architecture

A lot of definitions have been proposed and the definitions and the respective view on software architecture they represent have evolved over time. These definitions generally agree on several aspects that define the term *software architecture*. In the following, the most central definitions of software architecture found in literature are discussed and the aspects of software architecture they emphasize are described. A prominent definition of software architecture is given by Bass et al. [BCK12]. They define software architecture as follows:

Definition 2.1.1: Software Architecture by Bass et al. [BCK12]

The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

In their definition, the authors emphasize that the software architecture of a software system basically consists of multiple structures. While Bass et al. do not further define the types of structures, Rozanski and Woods distinguish between static and dynamic structures [RW11]. Static structures define design-time elements of the architecture such as modules, classes and packages, whereas dynamic structures define runtime elements and their dynamic interactions.

Another definition is given by the IEEE Standard 1471–2000 "Recommended Practice for Architectural Description of Software-Intensive Systems" [IEE00]:

Definition 2.1.2: Software Architecture by IEEE Standard 1471–2000

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

The previous definitions emphasize that software architecture provides a higher level of abstraction of the software system by intentionally focusing on the information that is necessary to reason about the system and to understand the system at a manageable level. Understanding and reasoning about a software system only on a code level is effort-intensive, since source code contains a lot of details that implement functional and non-functional requirements. Software architecture helps to abstract from these details, since information about details that are not relevant for reasoning about the system and understanding it are intentionally omitted. Viewing the software system from a higher level of abstraction supports to deal with the inherent complexity of the software system.

Another definition of software architecture is given by Taylor et al. [TVdH07]. In their point of view, software architecture is defined as:

Definition 2.1.3: Software Architecture by Taylor et al. [TVdH07]

The set of principle design decisions about the system.

Nothing is directly said here about the *fundamental structure* or *organization* of the system. Taylor et al. describe a design decision as something that *"encompasses every aspect of the system under development"*. The definition is consciously kept general. Design decisions may include decisions on the structural organization, on the functional behavior, on the interactions among elements, on the technology used, or even on the implementation. However, not every design decision is architecturally significant, i.e., has a *"wide impact on the structure and the quality attributes of the system."* [Kru04b]. This definition is a shift of the view on software architecture. Jan Bosch stated that *"rather than components and connectors, we need to model and represent a software architecture as the composition of a set of architectural design decisions"*. Since this shift, design decisions are considered as an important complement to the architecture design as stated by Kruchten et al. who proposed that *architecture knowledge = design decisions + design* [KLvV06]. This means that software architecture is not only considered as a design task, but also as a knowledge management task.

As can be observed, several definitions emphasizing different perspectives on software architecture exist. In this thesis, the understanding of software architecture is based on these definitions. Additionally, the perspective on software architecture is refined throughout the thesis, based on the existing definitions and based on the results of the empirical study presented in Chapter 3.

2.1.2. Roles of Software Architecture

Software architecture fulfills a multitude of roles during software development and evolution. In [Cle95], [Gar00], and [Smo02] several roles of software architecture are described. Generally, software architecture is the primary means of communication among stakeholders [Cle95]: it permits the analysis of functional and non-functional requirements before the realization of the software system [BCK12], and it supports project organization by being a work divider [PB01].

More specifically, Smolander defines the following four metaphors [Smo02] of software architecture in order to describe the general meaning of software architecture in software organizations:

Architecture as blueprint: The architecture and the related decisions constrain the detailed design and the system implementation. They are the input for the developers who need to realize the architecture and the decisions according to the given constraints.

Architecture as literature: The architecture is the central document for understanding the software system. It contains and preserves architecture knowledge, e.g. about decisions made in the past.

Architecture as language: In this metaphor, software architecture establishes a common language for communication between different stakeholders. The metaphor *Architecture as language* is also greatly advocated by Völter [Vö10] and will also be used throughout the thesis.

Architecture as decision: In this metaphor software architecture is perceived as the result of principal design decisions made during architecture design including the main drivers of the decisions and the rationale why a specific solution was chosen. Other authors also use this metaphor as for example Jansen et al. [JB05] and Taylor et al. [TVdH07].

The metaphors *Architecture as blueprint* and *Architecture as language* are of special importance during architecture enforcement and are the essential drivers of the approach developed in this dissertation. The main idea promoted in this thesis is to make the language explicit used to

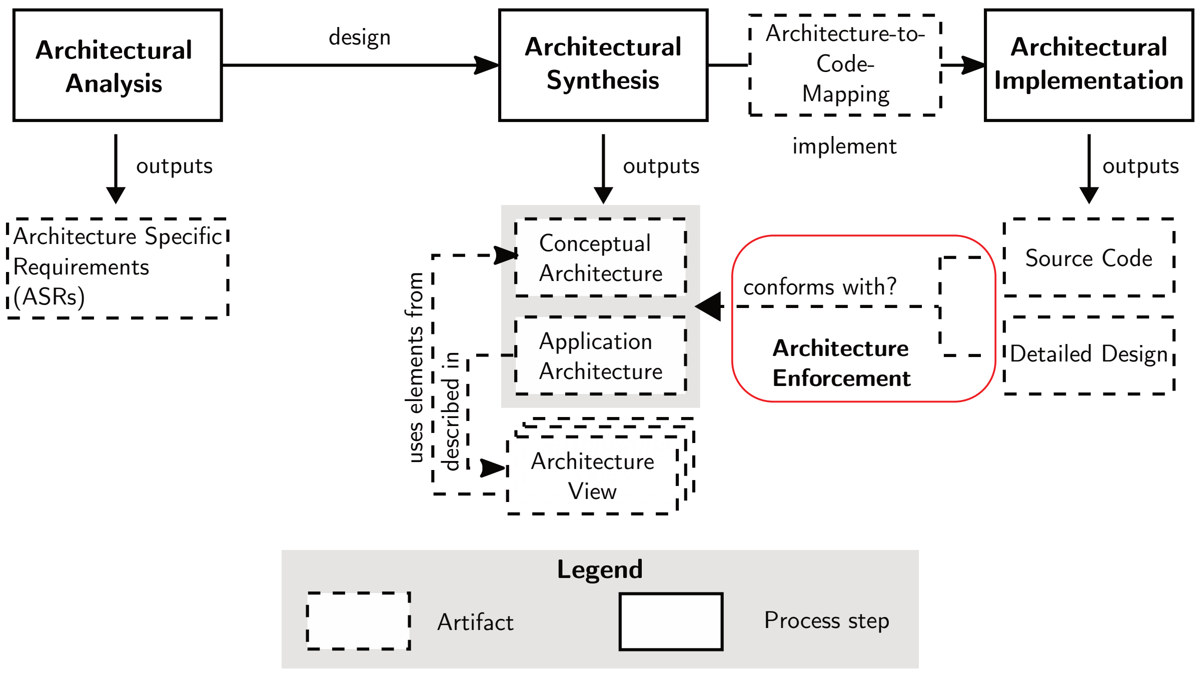


Figure 2.1.: *Main phases of the architecture-centric development process.*

describe and talk about the software architecture and to provide a flexible means to define and express this language. The approach therefore supports the *Architecture as Language* metaphor. The approach also supports the *Architecture as blueprint* metaphor by allowing the architect to enforce so-called *architecture rules* that can be expressed with the language.

2.1.3. Phases and Artifacts of Architecture-Centric Software Development

The architecture-centric software development process is divided into phases that are processed in iterations. Several authors propose different models in order to capture the most important activities performed during architecture design and the artifacts that are produced during each activity. Figure 2.1 visualizes the phases these models have in common, which artifacts they produce and how they are related with each other. Generally, the process contains the phases analysis [HKN$^+$07] [TAJ$^+$10] [WB12], synthesis [HKN$^+$07] [TAJ$^+$10] [WB12], evaluation [HKN$^+$07] [TAJ$^+$10] [WB12] (not shown in Figure 2.1), and architecture implementation [TAJ$^+$10] [WB12].

In the architectural analysis phase, the architect examines the architectural concerns. As a result, the software architect identifies a set of Architecture Significant Requirements (ASRs). In the architectural synthesis phase, the software architect creates suitable solutions, i.e., makes architectural design decisions, in order to address the ASRs. In this phase, the actual architecture (or several alternative architecture solutions) is created. Völter distinguishes two types of architectures that are produced in this step: the *conceptual architecture* and the *application architecture* [Völ05]:

Definition 2.1.4: Conceptual Architecture [Vö10]

The conceptual architecture defines the architectural elements used to describe the system on an architectural level and the relationships among these elements.

Definition 2.1.5: Application Architecture [Vö10]

The application architecture contains concrete instances of the elements defined in the conceptual architecture.

The conceptual architecture is independent of any technologies or platforms. It defines architectural building blocks, relationships, and constraints on how these building blocks are allowed to be related with each other. Architectural building blocks are called *architecture concepts*. For example, *components* or *interfaces* are architecture concepts. By defining the conceptual architecture, the software architects and developers – implicitly or consciously – decide for a language used to express the application architecture [Vö10]. As emphasized by Völter, it is of crucial importance to capture this language and to use it consistently throughout the development process [Vö10]. Having a consistent language about the software architecture enables stakeholders to effectively communicate about high-level structures [Smo02]. Ideally, the conceptual architecture is captured in a formal language in order to enable knowledge transfer, code generation, and automatic analyses [Vö10].

The application architecture uses concrete instances of the concepts defined by the conceptual architecture. The application architecture is documented and visualized in *architecture models* captured in multiple architecture views (see Section 2.2), e.g., static view, deployment view, or behavioral view. Each view describes a specific aspect of the software system. In these views, the architecture is described in terms of concepts provided by the conceptual architecture.

In the evaluation phase, the architect evaluates whether the proposed architectural solutions fulfills the ASRs or compares multiple solutions with each other and determines which alternative solution best fits the ASRs. The selected architectural solution is considered the *intended software architecture*:

Definition 2.1.6: Intended Software Architecture [DP09]

The intended software architecture is the architecture that exists in human minds or in the software documentation.

The architectural implementation phase realizes the selected architectural solution by creating a detailed design and the source code based on a suitable platform, i.e., the concrete technologies the architecture will be implemented with. This is, for example, the selection of a programming language, frameworks, or middleware. In order to be implemented, the architectural solution needs to be mapped to source code elements. For this, the software architect defines a so-called *architecture-to-code-mapping* (also called *platform mapping* or *technology mapping* [Völ05]).

Definition 2.1.7: Architecture-to-Code-Mapping

The architecture-to-code-mapping defines explicit rules on how elements of the conceptual architecture are mapped to the elements of a specific platform.

For example, the architects may decide that a *Component* must be mapped to a *Java package* that is named according to the concrete *Component* defined in the application architecture.

The distinction between platform-independent models and platform-specific models and the mapping between both is also followed by the model-driven architecture paradigm [KWWB03].

Consequently, the architectural implementation phase creates source code that implements the intended software architecture, i.e., it creates the *implemented software architecture*:

Definition 2.1.8: Implemented Software Architecture

The implemented software architecture is the software architecture that is manifested in the source code.

Since the synthesis phase and the implementation phase generate separate architectural artifacts on different abstraction levels, there is a high risk that both deviate from each other. That is why, it is necessary to ensure that the implementation still conforms to the architecture design created in the synthesis phase to ensure that ASRs are appropriately implemented. This is the goal of *architecture enforcement* (see Section 2.3).

In summary, two important aspects have been emphasized previously that also play an important role in this thesis:

- Firstly, the software architecture consists of the conceptual and the application architecture. Both should be captured explicitly in an architecture-centric development process in order to prevent knowledge vaporization and to enable tool-supported analyses. In the following sections, it is described which means are available to formally and informally describe software architectures (Section 2.2).
- Secondly, the architecture-centric development process produces separate architectural artifacts. These artifacts need to be consistent with each other in order to ensure that the implementation fulfills the requirements. In Section 2.3, architecture conformance checking is described as a method supporting to keep the implementation consistent with the software architecture.

In this thesis, an approach is developed that supports architecture enforcement. For this, the approach assumes the architecture-centric development process as described before. That is why, it is important to know about the main activities performed and artifacts created in this process. This especially means that an approach is needed that is able to (formally) express the conceptual architecture, the application architecture, and the mapping between the architecture and the implementation.

2.2. Modeling and Describing Software Architectures

Architecture documentation and descriptions are an important means for communicating the software architecture among stakeholders [CGB$^+$10], preserving architecture knowledge [Kru09] and for enabling further architectural analyses, such as architecture conformance checking and change impact analysis [JAvdV09]. Architecture descriptions are necessary to preserve the conceptual and application architecture in the software development process by capturing structural and behavioral properties of a software system. Different means for describing the conceptual and the application architecture exist. In the following, the fundamentals on software architecture specification and description are given.

2.2.1. Architecture Views and Viewpoints

The ISO/IEC 42010 standard on Systems and Software Engineering Architecture Description [IEE11] defines an architecture description as *"the collection of work products used to describe an architecture"*. An architecture description makes the software architecture explicit [Kru09] and consequently prevents architecture knowledge vaporization, i.e., loosing architecture knowledge [Bos04].

Ideally, architecture descriptions are organized in so-called *architecture views* [PW92] which allow for *"expressing different aspects of the architecture in an appropriate manner"*. The ISO/IEC 42010 standard Systems and Software Engineering - Architecture Description defines that an architecture description should be organized in multiple architecture views. In the standard, an architecture view is defined as *"the work product that represents the system from the perspective of architecture related concerns"* [IEE00]. Architecture views use concepts from the conceptual architecture to describe the application architecture (see Figure 2.1). Each view corresponds to an *architecture viewpoint*. An architecture viewpoint is *"a work product that establishes the conventions for the construction, interpretation and use of architecture views and associated architecture models"* [IEE00]. Since the standard does not prescribe which concrete viewpoints should be used, a lot of frameworks have been developed that propose viewpoints to be contained in an architecture description. For example, Kruchten proposed the 4+1 View Model of Software Architecture. It is part of the Rational Unified Process (RUP) and contains the logical view, the development view, the process view, and the physical view [Kru04b]. Hofmeister et al. have proposed the Siemens Four Views framework [HNS00]. Another viewpoint catalog has been developed by Clements et al. [CGB+10].

2.2.2. Modeling Languages for Architecture Descriptions

Architecture models associated with a viewpoint contained in an architecture description are expressed with *modeling languages*. In general, modeling languages differ in their level of formality. This means, architecture descriptions can be described with informal, semi-formal, or formal languages [TMD09]. When having a high level of formality, modeling languages allow for precise and unambiguous architecture descriptions as well as sophisticated architectural analyses. Formal modeling languages consist of a syntactic notation, i.e., syntax, and the meaning of the elements given in the syntax, i.e., semantics. Syntax elements can be words, sentences, statements, boxes, diagrams etc. Modeling languages can be graphical (e.g., diagrams), textual, or a combination of both [GKR+14]. Figure 2.2 summarizes a classification of modeling languages. It classifies exemplary modeling languages according to the dimensions "formality" and "notation". In the following, these exemplary modeling languages are presented.

Informal Modeling Languages

Informal architecture descriptions are mainly used for communicating the software architecture to the stakeholders. Informal descriptions are mostly written in natural language and sometimes enriched with informal "boxes-and-lines" diagrams [DLB14]. They cannot be processed automatically for further analyses, such as architectural evaluation or architecture conformance checking. Informal languages such as natural languages tend to be imprecise and ambiguous. This opens up the possibility of misinterpretation in the model. In such models, inconsistencies are often difficult to detect, since natural language is difficult to be processed automatically.

Notation				
	textual	Prose text	arc42 templates Architecture Decision Records Controlled Natural Language	Architecture Description Languages Domain Specific Languages Controlled Natural Language
	graphical	Boxes-and-Lines Diagrams	Unified Modeling Language	Architecture Description Languages Domain Specific Languages
		informal	**semi-formal**	**formal**
			Formality	

Figure 2.2.: *Examples of modeling languages and their classification according to the dimensions "formality" and "notation".*

This can lead to significant, probably non-obvious differences between design intent expressed in this model and its actual realization.

Semi-Formal Modeling Languages

Semi-formal modeling languages provide a defined syntax, but lack complete semantics definitions. They are mainly used to support design and documentation activities [Stö17], however they cannot – or only partially – be used for automatic architectural analyses due to the lack of semantics.

- The *Unified Modeling Language (UML)* is an example for a semi-formal architecture description language. Semi-formal descriptions can contain parts that are defined formally and informally. The syntax and semantics of the UML is formally defined by its metamodel that is part of the four-layer metamodel hierarchy of the Meta Object Facility [Obj06]. UML provides a multitude of diagrams that can be used for architecture descriptions, e.g., the component diagram or the deployment diagram. Diagrams can be annotated with explanations written in natural language. That is why, UML can be considered as a semi-formal language for describing architecture.

- Templates can be considered as a semi-formal way to describe software architectures. They provide named fields that can be filled with informal, natural text. The arc42 [HS] and *Architecture Decision Record (ADR)* templates are examples of template-based architecture documentations. For example, the ADR template suggested by Michael Nygard [adr] structures the documentation of an architecture design decision by the fields *title* (the addressed topic of the decision), *status* (capturing whether the decision has been made or is pending), *context* (the forces that drive the decision), *decision* (the response to the forces), and *consequences* (the resulting context after applying the decision).

- Controlled Natural Languages (CNLs) can be considered as semi-formal modeling languages that are able to bridge the gap between natural language and a formal language. It is worth to be noted that CNLs can also be designed as formal modeling languages in case they have a formally defined semantics, e.g., based on an existing logical formalism such as description logics. In Section 2.5 a detailed description on the foundations of CNLs is given. In this thesis, a formal CNL is developed to capture the conceptual architecture of a project and to define architecture rules.

Formal Modeling Languages

Formal modeling languages have a *"formally defined syntax and a semantics expressed in well-understood mathematical notation"* [BS01]. That is why those modeling language support comprehensive, automatic architecture analysis.

Architecture Description Languages (ADLs) are formally defined specification languages with a precise syntax and semantics [Cle96, MT00]. Due to their formal syntax and semantics, they allow for automatic architecture analyses. For example, the architecture description can be automatically analyzed for inconsistencies, e.g., as in [BEJV96] or [DHT05], or a partial implementation can be generated by means of model-driven development [Gar03]. Typically, ADLs are provided with a language infrastructure, e.g., an editor that supports to use the language for describing architecture or an analysis framework for architecture evaluations. ADLs describe the software architecture of a system using components and connectors. More precisely, the central elements of ADLs are the following [MT00]:

- *Components* are units of computations or data storage. A component has a port and properties. A port describes a point of interaction with other components via connectors.
- *Connectors* are means for the inter-component communication, i.e., they link components together.
- *Configurations* define the actual structure of the software architecture, i.e., which components a software architecture consists of and how they are connected with each other by connectors.
- *Composites* support the hierarchical composition of components. A composite component is described by a configuration of other components. It also has ports and properties which are mapped to ports and properties that link to internal components.

Due to their high level of formality, the following kinds of analysis are supported by ADLs:

- *Syntax Analysis*: The architecture description can be verified for syntactical correctness, since ADLs are modeling languages defined by a textual syntax.
- *Structural Analysis*: The architecture description can be checked for completeness and consistency.
- *Behavioral Analysis*: Some ADLs provide means to describe the behavior of software architecture, e.g., the interactions between components. Some ADLs therefore support the detection of deadlocks (e.g. Wright, [All97]) or schedulability analysis (e.g. MetaH, [BEJV96]).

ADLs have not found their way into mainstream software development [MR97, MLM$^+$13, LMM$^+$15, Ozk18b, Ozk18a] and are rather used in an academic context [WH05]. Due to the high cost of creating ADL-based architecture descriptions, ADLs are only used in a context where the costs for describing software systems provide sufficient value [TMD09]. This is especially true for software systems that are highly safety-critical [TMD09].

Modeling languages such as UML and ADLs are general-purpose languages for software architecture modeling. Since those languages provide generic and reusable architectural abstractions, they are not able to cover and define abstractions that are necessary for specific domains [Vö10]. Another aspect is that the meaning of architectural abstractions provided by such languages might not fit the project context. That is why, application-specific languages are needed that use concepts that are specific for the domain. Such a language is called a *Domain-Specific Language (DSL)*. These are formally defined languages that are tailored to a specific application domain [vDKV00]. They contain only the language elements that are necessary to describe architecture models for this application domain. DSLs can be divided into two subcategories according to the implementation method: external and internal [MHS05]. External DSLs are implemented from scratch, whereas internal DSLs are constructed by reusing the compiler or interpreter of a host language, e.g., Java. DSLs have a precise syntax and semantics and are therefore considered to be formal languages.

DSLs can be also built for architecture descriptions. In a case study, Völter develops a DSL for describing the software architecture [Vö10]. The main idea is to build an architecture-specific DSL that *captures the core architectural abstraction of the particular architecture* and to use the DSL to generate the corresponding code in order to implement it consistently. The main advantage is – as opposed to UML and ADLs – that software architects are not restricted to predefined architectural abstractions. This means, the DSL is designed for representing the architectural abstractions as needed to describe the application architecture. The abstract syntax of the language contains all the necessary, project-specific constructs to express the conceptual architecture (see Section 2.1).

2.3. Architecture Enforcement

The general model as presented in Section 2.1 reveals the different phases performed during the architecture-centric development process. As elaborated in Section 2.1, the software architect's duty in this process is to make architectural design decisions and to create a suitable architecture design that is eventually used to guide the architectural implementation phase. Another challenging duty of the software architect – as emphasized in Figure 2.1 – is *architecture enforcement*. Generally, architecture enforcement is defined as

Definition 2.3.1: Architecture Enforcement [Zim09]

Architecture enforcement is concerned with the correct and seamless implementation of architectural design decisions.

Architecture enforcement is necessary in order to ensure that the software system fulfills the ASRs identified in the architectural analysis phase. Architecture enforcement is also suggested as an important activity by state-of-the-art development processes. For example, RUP [Kru00] advises software architects to enforce the architecture design by refining it in small and actionable increments. In agile processes (e.g., Scrum [SB01]), architects do not have a dedicated role. However, agile processes strongly advocate the importance of face-to-face communications to avoid misunderstandings and to find consensus in architecture design and implementation.

Generally, architecture enforcement has two goals **[Sch18b]**:

1) Ensure the agreement on design decisions with stakeholders: A software architect needs to share his design decisions with stakeholders, e.g., developers, and needs to make sure that they accept and understand the decisions before starting with the implementation. This agreement is important in order to minimize problems during the implementation of design decisions. For example, if developers are not familiar with certain patterns or technologies, they might not be able to realize design decisions. To achieve this goal, an architect needs to describe a road-map for the implementation of a system, which includes *fundamental guidelines* developers should follow during implementation.

2) Check the conformance between implementation and design decisions: An architect needs to ensure the correct implementation of design decisions during the software development life cycle. In other words, he has to make sure that the developers follow the implementation road-map and guidelines [BCK12]. Derivations from the design decisions might occur for several reasons, knowingly or unknowingly. For example, strict deadlines for an additional feature might force developers to purposefully disobey the guidelines or developers might misunderstand them and unintentionally violate the guidelines.

2.3.1. Architecture Erosion

The definitions presented in Section 2.1 not only focus on the organization of the system. They emphasize that the software architecture prescribes principles and guidelines on how the software system is allowed to be designed and evolved. Software architecture therefore serves as a constraining blueprint on subsequent refinement phases, such as design and implementation. This perspective is also emphasized by Fairbanks [FG10]. He states that the software architecture impacts and constrains the way how the code is allowed to be or not allowed to be implemented and to evolve. Guiding principles are part of the software architecture that may have implications on the implementation of the software architecture. In the context of this thesis, these "constraining blueprints" and "guiding principles" are considered as *architecture rules*:

Definition 2.3.2: Architecture Rule

An architecture rule is prescribed by the software architecture and constrains the implementation of a software system.

For example, architecture patterns such as the Layered Pattern [BMR$^+$96] imply that there are system parts that are only allowed to interact in a specific way, e.g., lower layers are not allowed to call upper layers. This has implications for the implementation in the sense that classes implementing functionality of a lower layer are not allowed to use classes implementing functionality of an upper layer. Consequently, software architecture restricts the way how the detailed design and the implementation can be refined and which design decisions can be made during implementation.

If the architect misses to appropriately define and enforce architecture rules, in order to address the goals of enforcement, there is a high risk that the implementation violates prescribed architecture rules. As a result, the intended and implemented architecture deviate from each other. In this thesis, the deviation is considered as *architecture erosion*:

> **Definition 2.3.3: Architecture Erosion [TMD09]**
>
> Architecture erosion is the introduction of architectural design decisions into a system's descriptive architecture that violate the prescriptive architecture.

As a consequence, erosion causes a gradual regression for the quality of a software system (e.g., lower maintainability). As defined in Section 2.1, software architecture is an abstraction of the software system that allows for reasoning about system properties, e.g., quality attributes. Only when the implemented architecture aligns with the intended architecture, the software architecture can be used as a valuable predictive and descriptive abstraction of a software system. If the implementation is not built according to what was planned, reasoning about risks, qualities, and other aspects cannot be performed efficiently. Additionally, software architecture as a means for communication and comprehending the software system becomes unreliable, since it is unclear to which degree the architecture is represented in the implementation. That is why, it is necessary to constantly verify whether software architecture and implementation still align with each other, i.e., to avoid or to at least minimize architecture erosion. As a means for detecting architecture erosion and for supporting architecture enforcement, architecture conformance checking provides a useful method. This method is presented in the following section.

2.3.2. Architecture Conformance Checking

Architecture erosion is a problem that occurs slowly over time [Mer09]. Therefore, architects need to continuously check the adherence of the implementation to the architecture in order to support the early detection of architecture erosion. This process is called *architecture conformance checking* [KN16, Her11, CLN15].

In this section, the terminology of conformance checking and corresponding definitions as they are used in this thesis are introduced and explained.

Architecture conformance checking provides a useful means for detecting and controlling architecture erosion [DSB12]. Architecture conformance checking (sometimes referred to as architecture compliance checking [KN16]) aims for revealing the differences between the software architecture and its implementation.

Knodel defines architecture conformance checking as follows

> **Definition 2.3.4: Architecture Conformance Checking [KN16]**
>
> Architecture conformance checking is a technique that verifies to which degree the implemented architecture conforms with the intended architecture.

The goal of architecture conformance checking is to detect *architecture violations* in order to investigate to which extent an implementation conforms with the software architecture:

Definition 2.3.5: Architecture Violation [KN16]

An architecture violation is an architectural element or an architectural relationship between elements that has a counterpart in the implementation which is not realized as specified.

The accumulation of architecture violations is considered as architecture erosion. An implementation that does not contain any architecture violations is conform to the specified architecture. Architecture conformance is then defined as follows:

Definition 2.3.6: Architecture Conformance [DKL09]

Architecture conformance captures the degree of having accomplished required or requested demands realized in the implementations of software systems. Architecture conformance means that the specified architecture is equivalent to the implemented architecture.

Architecture erosion can therefore also be considered as the *lack of conformance* [RLGBAB08] [LV95] or the *lack of architecture compliance* [vHRH$^+$09]. The process of architecture conformance checking is performed by the following steps [KN16]:

Specifying the Intended Architecture: The intended architecture specifies the aspects that need to be checked for conformance. It describes how the software architecture should be realized in the source code. The intended architecture can be described with graphical architecture models or as a collection of architecture rules. This depends on the method chosen for architecture conformance checking (see Chapter 4). Depending on the scope of architecture conformance checking, a specific view is expected. For example, if the static structure in terms of modules and dependencies between the module should be verified, the module view is expected as an input [DKL09]. In case, the communication of components during runtime is of interest, a component-and-connector-diagram is used to specify the intended architecture [Nic18]

Source Code Fact Extraction: In this step, relevant facts from the source code are extracted using reverse engineering techniques. As mentioned by Knodel et al. the fact extraction should focus only on the aspects that are necessary to perform the conformance checking, since the reverse engineering effort becomes exhaustive in terms of time and memory consumption [KN16]. For example, the class and package structure is extracted from the source code. In case the runtime behavior of the system should be investigated, runtime traces can be extracted.

Discovering the Implemented Architecture: Due to the conceptual gap between software architecture and the implementation, the implemented architecture is not directly visible from the source code [KN16]. That is why, the implemented architecture first needs to be discovered from the source code. Based on the extracted facts from the previous step, architectural elements are discovered from lower level source code entities. This aligns the architecture and source code model to the same level of abstraction, i.e., *lifting* the source code model onto the architectural level. This is accomplished by mapping source code entities to their counterparts in the architecture model, i.e., architecture-to-code-mapping

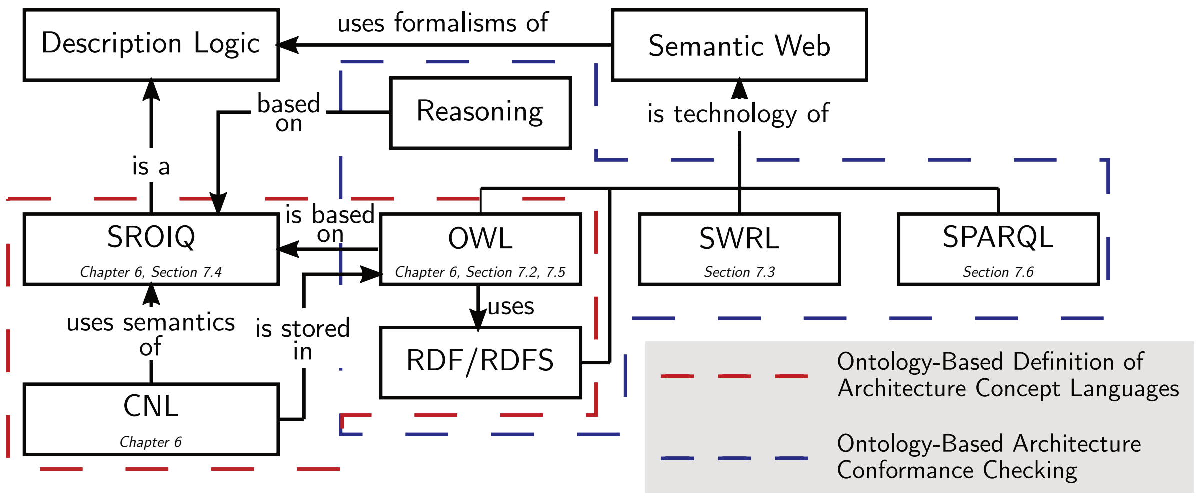


Figure 2.3.: *Overview on applied formalisms and technologies, their relations, and the mapping to the respective part of the developed approach.*

is performed (see Section 2.1). For example, classes are mapped to a specific architecture component. This can be performed manually or automatically. The result of the mapping process reveals the implemented architecture.

Analysis: In the actual analysis phase, the differences, i.e., architecture violations, between the intended architecture and the implemented architecture are calculated.

Interpretation: The detected violations are analyzed, for example according to their severity. Based on the interpretation, required repair actions are planned.

2.4. Description Logics and Ontologies

The following sections describe the fundamentals of description logics, ontologies and CNLs. This background information is essential to understand how the proposed approach is implemented using these formalisms. Figure 2.3 provides an overview on the applied formalisms and technologies and the mapping to the corresponding part of the approach where they have been applied.

2.4.1. Description Logics

Description logics are a family of the logic-based knowledge presentation formalisms [BCM$^+$03]. They have a formally defined syntax and semantics and therefore allow for a precise specification of concepts and their properties in a domain of interest. They are widely used in ontological modeling [BHS05]. Due to their formality, they allow for logical deduction, i.e., to infer additional knowledge from explicitly stated information. This distinguishes description logics from other modeling languages such as UML [AGK06] [KSH12] (see Section 2.2.2).

Different description logics languages exist that provide distinct levels of expressiveness. Each language is defined by allowing or disallowing specific constructs. The more constructs are allowed, the more expressive the language is. This impacts the reasoning complexity, i.e., the complexity to derive conclusions from existing knowledge. In this dissertation, the description language $\mathcal{SROIQ}$ [HKS06] is used. It constitutes one of the most expressive description language with the advantage that reasoning is still decidable [HKS06]. The following explanations always focus on the formal syntax and semantics of this language.

Regardless of the language used, description logics formalize an application domain by *concepts*, *roles*, and *individuals*. In the following, N_C, N_R, and N_I denote the disjoint set of concept names, role names, and individual names, respectively. The triple (N_C, N_R, N_I) is called the *signature* or also the *vocabulary* [BCM+03]:

- Concepts are the central entities in this formalism. They represent sets of objects, classes of entities, or categories characterized by common properties. They roughly correspond to unary relations.

- Roles are relations between concepts. They can be thought of as binary relations. They denote semantically meaningful associations between concepts.

- Individuals denote singular entities. They represent constants in the formalism.

Concept names, role names, and individual names can be chosen arbitrarily, i.e., one is not restricted by a predefined set of names for concepts, roles, and individuals.

A description-logics knowledge base consists of three parts, namely the *terminological box* (TBox), the *relational box* (RBox) and the *assertional box* (ABox). The knowledge base is also called *ontology* [BCM+03] [KSH12].

The TBox describes relationships between concepts. It contains the so-called *general concept inclusion (GCI) axioms* (also called subsumption axioms, class axioms or axioms for short). Those have the form $C \sqsubseteq D$ where C and D are concept names from N_C. This axiom type can be thought of as a *is-a* relationship implying a hierarchical relationship between concepts (everything that is a C must also be a D). For example, the GCI axiom $Module \sqsubseteq ArchitecturalElement$ states that all modules are considered as architectural elements.

RBox axioms define the properties of roles and relationships between roles. Axioms in the RBox are called *role inclusion axioms*. For example, the axiom $partOf \sqsubseteq dependsOn$ defines that *partOf* is a sub-role of *dependsOn*. Every individual that is related with another individual by *partOf* is also related with this individual via the relationship *dependsOn*. In a role inclusion, role composition allows for a more complex role definition. For example, the role inclusion axiom $commits \circ modifyFile \sqsubseteq isAuthorOf$ states the following: if a developer commits a change and the change modifies a file, then the developer is an author of the file.

The ABox contains information about single individuals of a domain. An individual assertion can have the form $C(a)$ (concept assertion) or $r(a,b)$ (role assertion) where $a,b, \in N_I$ are individual names, C is a concept and r is a role. For example, the concept assertion $Module(gui)$ asserts that *gui* is a module, more precisely that the individual named *gui* is an instance of the concept $Module$. The exemplary role assertion $dependsOn(gui, logic)$ asserts that the module *gui* depends on the module *logic* (assuming *logic* is asserted as $Module(logic)$), more precisely that the individual named *gui* is in the relation that is represented by *dependsOn* to the individual named *logic*.

It is important to note that *gui* and *logic* are considered as different individuals in the previous examples. However, description logics do not make the *Unique Name Assumption (UNA)* [KSH12] meaning that different names might refer to the same individual. This means, the individuals *gui* and *logic* are identical. If needed otherwise, this must be stated explicitly. This is called an *individual inequality assertion*. For example, the statement $gui \not\approx logic$ specifies that *gui* and *logic* are different individuals.

2.4.2. Syntax of SROIQ

Concept Constructors and Role Restrictions: In the previous section, it was shown how to define new concepts and how to relate concepts with each other by GCIs. However, this kind of modeling is quite restrictive and limited for comprehensive ontological modeling. Therefore, description logics allow new concepts and roles to be described by using constructors. With constructors, concepts can be defined by *concept descriptions*. $SROIQ$ allows to specify concept descriptions such as universal restriction ($\forall$), existential restriction ($\exists$), and qualified number restrictions ($= n$,$\leq n$,$\geq n$, where n is a natural number) in order to allow for an accurate and comprehensive ontological modeling. The syntax of $SROIQ$ is defined as follows:

Definition 2.4.1: SROIQ Syntax [KSH12]

Given a set N_I of individual names, a set N_C of concept names, and a set N_R of role names. Then the set of *role expressions* $\mathbf{R}$ is defined by the grammar:

$$\mathbf{R} ::= U | N_R | N_R^-$$

where U is the universal role and N_R^- is the set of inverse roles. Then the set of SROIQ *concept descriptions* $\mathbf{C}$ is defined as follows:

$$\mathbf{C} ::= N_C | \mathbf{C} \sqcap \mathbf{C} | \mathbf{C} \sqcup \mathbf{C} | \neg \mathbf{C} | \top | \bot | \exists \mathbf{R}.\mathbf{C} | \forall \mathbf{R}.\mathbf{C} | \geq n\mathbf{R}.\mathbf{C} | \leq n\mathbf{R}.\mathbf{C} | \exists \mathbf{R}.Self | \{N_I\}$$

where n is a natural number.

The definition describes that concept descriptions are inductively defined as follows

- Basic concepts descriptions can be formed according to the following syntactical rules:
  - $\top$ and $\bot$ are concept descriptions, called the *top concept* and the *bottom concept*, respectively.
  - Every $A \in N_C$ is an atomic concept. If C and D are concepts, then so are $\neg C$ (negation), $C \sqcap D$ (intersection), and $C \sqcup D$ (union).
- Let $R \in N_R$ be a role, $\exists R.C$ (existential restriction) is a concept, and $\forall R.C$ (universal restriction) is also a concept. Given $n \in \mathbb{N}$, then $\exists R.Self$ (self restriction), $\geq nR.C$ (at-least restriction), and $\leq nR.C$ (at-most restriction) are also concepts. These restrictions are called *role restrictions*.

The intersection of two concepts ($C \sqcap D$) represents the set of individuals that belong to both concepts C and D. For example, the concept description $Module \sqcap External$ represents the individuals that are external modules (e.g., external libraries).

The union $C \sqcup D$ is the dual of intersection and represents the individuals that either belong to C, D or both concepts. For example, the concept description $Developer \sqcup Committer$ describes individuals that are either developers, committers, or both.

The negation can be used to describe individuals that do not belong to a concept. For example, individuals that are modules but are not external libraries can be modeled as $Module \sqcap \neg External$.

Using the bottom concept and the intersection it can be stated that two concepts are disjoint meaning that individuals cannot belong to those two concepts at the same time, namely by defining the axiom $C \sqcap D \sqsubseteq \bot$. For example, the axiom $Module \sqcap Interface \sqsubseteq \bot$ states that no individual can be both a module and an interface.

With role restrictions, the relationships between concepts and roles can be described. Definition 2.4.1 describes the existential restriction, universal restriction, self restriction, and the cardinality restriction.

With the existential restriction ($\exists R.C$), the set of individuals can be described that are related with at least one individual via a specific role. For example, the concept description $\exists defines.Interface$ defines the set of individuals that define at least one interface. This concept description can be used as a super concept in a GCI axiom. For example, the axiom $Module \sqsubseteq \exists defines.Interface$ describes if an individual of the concept $Module$ has a relation $defines$ to an individual than at least one of these individuals must belong to the concept $interface$.

The universal restriction ($\forall R.C$) describes individuals that are connected to individuals belonging to the concept C only by the role R. Using the previous example with a universal restriction yields: $Module \sqsubseteq \forall defines.Interface$. This means that all individuals belonging to the $Module$ concept and participating in the role $defines$ can only be connected to individuals belonging to the $Interface$ concept via this role. This axiom defines a constraint stating that modules can only define interfaces and no other elements.

Universal and existential restrictions can be used to define domain and range restrictions, i.e., restrictions on the concepts that can be in the domain and the range of a role. The domain of a role is restricted by the axiom $\exists R.\top \sqsubseteq D$, where D defines the domain of the role R. The range of a role is restricted by the axiom $\top \sqsubseteq \forall R.D$, where D is the range of the role R.

The set of GCI axioms defines an *ontology*. More specifically, by describing axioms using the SROIQ description language, this ontology is called the *SROIQ ontology*:

Definition 2.4.2: SROIQ Ontology [KSH12]

Axioms of the ABox, TBox, and RBox have the following form [KSH12]:

- ABox: $\mathbf{C}(N_I)$, $\mathbf{R}(N_I, N_I)$, $N_I \approx N_I$, $N_I \not\approx N_I$
- TBox: $\mathbf{C} \sqsubseteq \mathbf{C}$, $\mathbf{C} \equiv \mathbf{C}$
- RBox: $\mathbf{R} \sqsubseteq \mathbf{R}$, $\mathbf{R} \equiv \mathbf{R}$, $\mathbf{R} \circ \mathbf{R} \sqsubseteq \mathbf{R}$, $Disjoint(\mathbf{R}, \mathbf{R})$

The axioms $N_I \approx N_I$ and $N_I \not\approx N_I$ define an equality relation between individuals. The $\equiv$ relation defines that two concept (or role) descriptions are equivalent.

The set of such axioms is called the *knowledge base* or the *SROIQ ontology* [KSH12] written as $\mathcal{KB}$.

Role Properties: The description language SROIQ allows the definition of inverse roles. For example, the equivalence $contains \equiv isContainedIn^-$ describes that if a module contains a package then this package is contained in this module. The role $isContainedIn^{-}$ is the inverse role of $isContainedIn$.

Roles can be described by various characteristics such as role transitivity, symmetry, asymmetry, reflexivity, and irreflexivity. For example, the role *contains* is transitive, e.g. if a module m_1 contains a module m_2 that contains another module m_3, then m_1 also contains m_3.

2.4.3. Description Logics Semantics

In order to make conclusions about stated knowledge in form of logical consequences, the formal meaning of axioms needs to be defined. The semantics define the meaning of the concept, role, and individual names that are used, i.e., establish a formal relationship between the symbols used and the objects they actually represent. For example, in the previous section, identifiers such as *gui* (individual), *Module* (concept), *dependsOn* (relation) and *Interface* (concept) are used. These are only syntactic identifiers with no meaning associated.

An *interpretation* determines the semantic counterparts of the vocabulary elements specified in N_C, N_R, and N_I. An interpretation $\mathcal{I}$ consists of a set $\Delta^{\mathcal{I}}$ called the domain of $\mathcal{I}$ (or universe of discourse) [KSH12]. This can be understood as the entirety of things existing in the world that $\mathcal{I}$ represents. Additionally, an *interpretation function* maps each atomic concept A to a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, each atomic role R to a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, and each individual name a to an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. In this way, the interpretation describes the meaning of all entities.

Example 2.4.1: Interpretation

Consider the following signature:

- $N_I = \{business, gui, data\}$
- $N_C = \{Module\}$
- $N_R = \{dependsOn\}$

An interpretation $\mathcal{I} = (\delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ can be defined as follows:

- $gui^I = \{\circ\}$, $business^I = \{\Box\}$, $data^I = \{\Diamond\}$
- $Module^I = \{\circ, \Box, \Diamond\}$
- $dependsOn^I = \{\langle\circ, \Box\rangle,\ \langle\Box, \Diamond\rangle\}$

It is also necessary to define semantics of concepts and roles. The semantics of a concept description can be obtained from the semantics of its constituents (compositional semantics). Table 2.1 summarizes the syntax and semantics of the $\mathcal{SROIQ}$ constructors, where Table 2.2 depicts the syntax and semantics of axioms for the ABox, TBox, and RBox.

The purpose of the interpretation function is to determine the satisfaction of axioms. If an axiom α is true given a specific interpretation $\mathcal{I}$, one says that $\mathcal{I}$ is a *model* of α or that $\mathcal{I}$ satisfies α, written $\mathcal{I} \vDash \alpha$.

Table 2.1.: *Syntax and semantics of the SROIQ constructors. Adopted from [HST99].* $a \in N_i$ *is an individual name,* $A \in N_c$ *is a concept name,* C, D *are concepts,* R *is a role.*

	Syntax	**Semantics**
Individuals		
individual name	a	$a^{\mathcal{I}}$
Roles		
atomic role	R	$R^{\mathcal{I}}$
inverse role	R^-	$\{\langle x,y\rangle \mid \langle y,x\rangle \in R^{\mathcal{I}}\}$
universal role	U	$\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
Concepts		
atomic concept	A	$A^{\mathcal{I}}$
intersection	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
union	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
complement	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
top concept	$\top$	$\Delta^{\mathcal{I}}$
bottom concept	$\bot$	$\emptyset$
existential restriction	$\exists R.C$	$\{x \mid \exists y.\langle x,y\rangle \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\}$
universal restriction	$\forall R.C$	$\{x \mid \forall y.\langle x,y\rangle \in R^{\mathcal{I}} \text{ implies } y \in C^{\mathcal{I}}\}$
at-least restriction	$\geq nR.C$	$\{x \mid \#\{y.\langle x,y\rangle \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\} \geq n\}$
at-most restriction	$\leq nR.C$	$\{x \mid \#\{y.\langle x,y\rangle \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\} \leq n\}$
local reflexivity	$\exists R.Self$	$\{x \mid \langle x,x\rangle \in R^{\mathcal{I}}\}$

Example 2.4.2: Satisfaction of Axioms

Consider the interpretation $\mathcal{I}$ of Example 2.4.1. The exemplary axiom $dependsOn(business, gui)$ does not hold in $\mathcal{I}$, since the pair of the respective individuals is not contained in the extension of the *dependsOn* role:

$$\langle business^I,\ gui^I\rangle = \langle \square, \circ\rangle \notin dependsOn^I$$

This can be extended to the whole knowledge base: $\mathcal{I}$ is a *model* of a given knowledge base $\mathcal{KB}$ ($\mathcal{I}$ satisfies $\mathcal{KB}$, written $\mathcal{I} \vDash \mathcal{KB}$) if it satisfies all the axioms of $\mathcal{KB}$, i.e., if $\mathcal{I} \vDash \alpha$ for every $\alpha \in \mathcal{KB}$. A knowledge base $\mathcal{KB}$ is called *satisfiable* or *consistent* if it has a model, and it is called *unsatisfiable* or *inconsistent* or *contradictory* otherwise.

Formal semantics are a prerequisite for reasoning support. In the next section, typical reasoning tasks – as they are applied to implement the approach of this thesis – are described.

2.4.4. Reasoning Tasks

Once a knowledge base has been defined and described, this knowledge base can be queried and new knowledge can be derived from it. For this, various reasoning tasks can be performed on the knowledge base. Basic inference problems are:

- *Satisfiability of TBox*: Verifies whether a concept C is satisfiable with respect to a

Table 2.2.: *Syntax and semantics of the SROIQ axioms. Adopted from [KSH12].*

	Syntax	**Semantics**
ABox		
concept assertion	$C(a)$	$a^{\mathcal{I}} \in C^{\mathcal{I}}$
role assertion	$R(a,b)$	$\langle a^{\mathcal{I}}, b^{\mathcal{I}} \rangle \in R^{\mathcal{I}}$
individual equality	$a \approx b$	$a^{\mathcal{I}} = b^{\mathcal{I}}$
individual inequality	$a \not\approx b$	$a^{\mathcal{I}} \neq b^{\mathcal{I}}$
TBox		
concept inclusion	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
concept equivalence	$C \equiv D$	$C^{\mathcal{I}} = D^{\mathcal{I}}$
RBox		
role inclusion	$R \sqsubseteq S$	$R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$
role equivalence	$R \equiv S$	$R^{\mathcal{I}} = S^{\mathcal{I}}$
complex role inclusion	$R_1 \circ R_2 \sqsubseteq S$	$R_1^{\mathcal{I}} \circ R_2^{\mathcal{I}} \subseteq S^{\mathcal{I}}$
role disjointness	$Disjoint(R,S)$	$R^{\mathcal{I}} \cap S^{\mathcal{I}} = \emptyset$

knowledge base. Satisfiability checking is useful in order to verify whether an ontology is meaningful, i.e., whether it contains no contradictory statements. An inconsistent knowledge base often hints to modeling errors. That is why, deciding whether a knowledge base is consistent is an important task in order to build ontologies of high quality. Formally, given a knowledge base $\mathcal{KB}$ a concept $C \in \mathbf{C}$ is called satisfiable with respect to $\mathcal{KB}$ if it may contain individuals, i.e. there is a model $\mathcal{I}$ of $\mathcal{KB}$ that maps C to a nonempty set ($C^{\mathcal{I}} \neq \emptyset$).

- *Subsumption*: Subsumption hierarchically organizes concepts according to their generality. Subsumption checking verifies whether the concept C_{sub} is subsumed by the concept C_{sup}. C_{sub} is subsumed by C_{sup} if $C_{sub}^{\mathcal{I}} \subseteq C_{sup}^{\mathcal{I}}$ for all models $\mathcal{I}$ of the TBox and the RBox.
- *Consistency (ABox)*: Proves whether the ABox is consistent with regard to the TBox and RBox (the ABox is consistent, if it has a model $\mathcal{I}$ which is also a model of the TBox and RBox).
- *Realization*: Computes the *instance-of* relationship for every individual. It finds the most specific concept for the respective individual it belongs to.

Highly optimized reasoning algorithms have been developed and have shown that tableau algorithms, even for highly expressive description logics, lead to a good performance even on large knowledge bases [BCM+03]. Reasoning services allow for an automatic deduction of implicit knowledge from explicitly stated knowledge. Reasoning on description-logic knowledge bases is decidable and therefore always yields a correct answer in finite time. The majority of state-of-the art reasoners such as Pellet, FaCT++, or RacerPro use tableau methods [DR99].

2.4.5. The Web Ontology Language

The Web Ontology Language (OWL) [owl12a] is a formal language for describing ontologies. It is built upon the Resource Description Framework (RDF) [KC04] and RDF Schema (RDFS) [BG04]. Together, they constitute main standards and technologies of the *Semantic Web* to describe structured data and ontologies. The semantic web is an extension of the World Wide Web [HKR09]. Using standards like RDF, RDFS, and OWL as common data formats enable to represent, share, and integrate information between heterogeneous applications and systems. Additionally, these standards promote for enriching data with additional meaning that can be understood by machines in order to intelligently search, combine, and process Web content.

RDF and RDFS use labeled, directed graphs as the data model to represent objects and their relations. Objects are nodes in the graph and directed edges in the graph are the relations between these objects. RDF is domain independent in a sense that it does not make any assumptions about a particular domain. Objects are called *resources* and are uniquely identified by URIs. For example, a resource is a person, a book, or a web page that can be identified by such a URI.

The graph-based data model formalizes data based on subject-predicate-object triples that are called *RDF statements*. An RDF triple is used to make a statement about a resource of the real world. They assert the property, i.e., the relationship of two resources. For example, the person "John Doe" (subject) is author of (predicate) the book "Semantic Web for Beginners" (object). The object can be another resource or a literal. A literal is an atomic value such as string or integer. An RDF graph is a set of statements. RDF provides predefined predicates in order to define resources and properties. The predicate *type* is a relation predefined by RDF in order to indicate whether an RDF element is a resource or a property.

With RDF, it is not possible to formally define terms of a knowledge domain. However, in the context of Semantic Web, this becomes necessary in order to allow for logical inferences. With RDFS, *classes* can be defined explicitly (using the predicate *Class*) and inheritance between classes can be defined (using the predicate *subClassOf*). Consequently, RDFS provides a richer vocabulary than RDF allowing for a definition of light-weight ontologies. Figure 2.4 visualizes the elements of RDF and RDFS and their correspondences. The prefixes *rdf* and *rdfs* are variables holding the namespaces – identified by URIs – of RDF and RDFS in which the corresponding predicates are defined. These prefixes are conventionally used for these namespaces. For example, *rdf:Resource* means that the predicate *Resource* is part of the RDF vocabulary. Figure 2.4 writes out the entire namespaces as defined by W3C for RDF [W3Ca] and for RDFS [W3Cb].

Figure 2.5 shows an example of an RDF/RDFS graph. Subject and object elements are depicted as ellipses, whereas predicates are visualized as edges between ellipses. The example defines the two classes *Person* and *Book*. The resource *johndoe* is an instance of the class *Person*. This is denoted by the predefined relation *rdf:type*. Additionally, the ontology defines a *hasName* property. This property assigns an instance of a *Person* or of a *Book* to a string value denoting a name.

OWL [owl12a] is an extension of RDF and RDFS. OWL reuses all the elements defined by RDF and RDFS, however, it provides a richer vocabulary than these languages. In this way, more comprehensive ontologies can be defined with OWL. More concretely, OWL supports all constructs of the $\mathcal{SROIQ}$ formalism described in Section 2.4.2. Furthermore, it provides some additional features, such as ontology versioning information, annotations, and modeling and

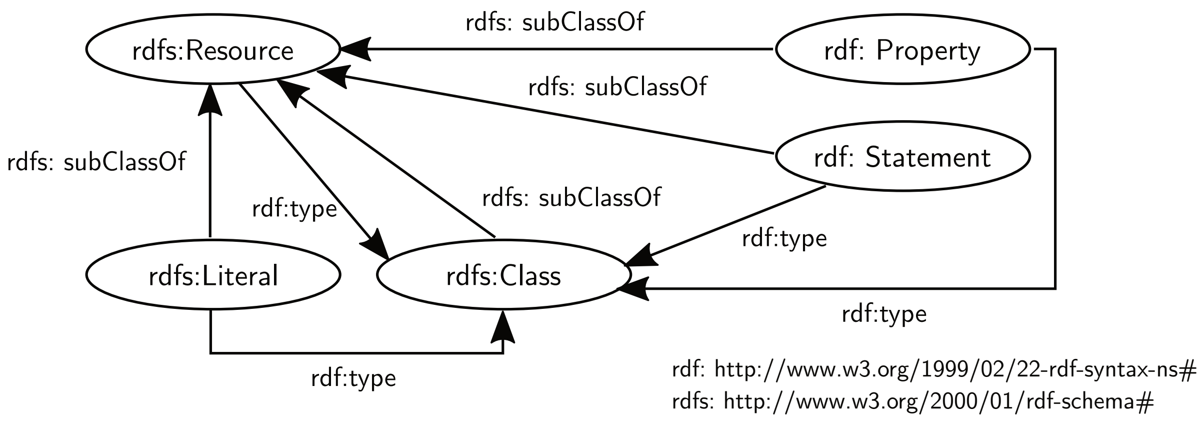


Figure 2.4.: *Central constructs of RDF and RDFS and correspondences.*

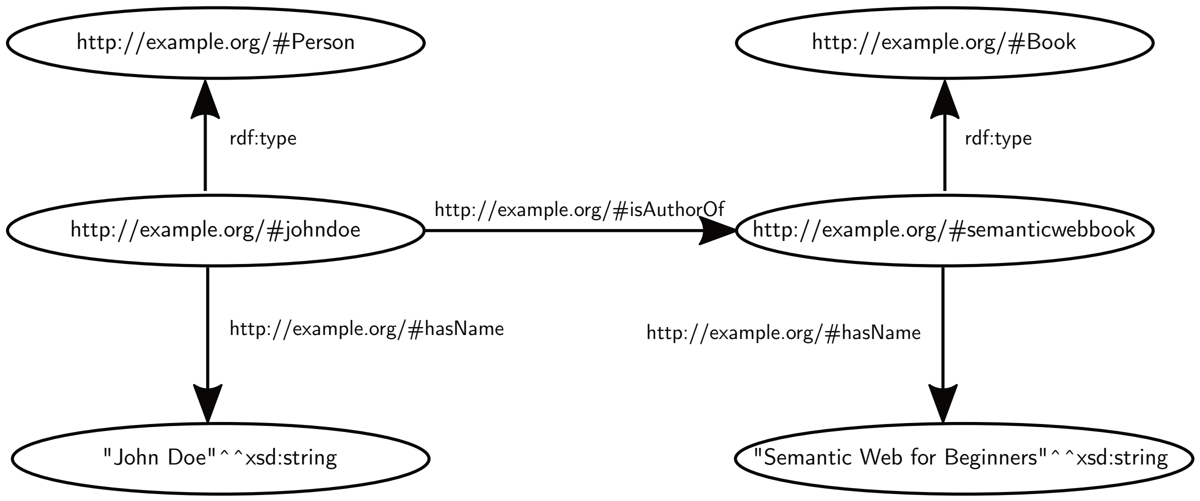

http://example.org/#Person
http://example.org/#Book
rdf:type
rdf:type
http://example.org/#johndoe
http://example.org/#isAuthorOf
http://example.org/#semanticwebbook
http://example.org/#hasName
http://example.org/#hasName
"John Doe"^^xsd:string
"Semantic Web for Beginners"^^xsd:string

Figure 2.5.: *Example of an RDF/RDFS graph.*

Table 2.3.: Comparison of the terminology of description logics and OWL.

Description Logics	OWL
concept name	class name
concept	class
concept description	class expression
role name	object property name
role	object property
knowledge base, ontology	OWL ontology
(GCI, class) axiom	(class) axiom
vocabulary / signature	vocabulary

reasoning with data types.

While the building blocks of OWL are very similar to that of description logics, they differ in their terminology, e.g., concepts are called *classes* and roles are called *properties.* A *property* in OWL corresponds to a role in description logics and therefore describes a relationship. Further, OWL distinguishes between two types of properties, namely *object properties* and *data type properties.* The former relates individuals to other individuals, while the latter relates individuals to literal values of a certain datatype (e.g., integer, double, string). The correspondences between OWL and description logics terminologies are depicted in Table 2.3.

Table 2.4 depicts the correspondences between the OWL and description logics syntax. Since description logics are the formal basis of OWL, syntactical elements of OWL syntax can be mapped correspondingly to syntactical elements of description logics. For describing OWL constructs, the functional-style syntax [owl12b] is used. Besides the functional-style syntax, there exist many other ways for saving and sharing OWL ontologies, as for example RDF/XML syntax [rdf14], OWL/XML syntax [owl03], Manchester syntax [man12], and Turtle [tur08]. In this thesis, the functional-style syntax is used, since it allows ontologies to be written in a more compact form compared to the other syntaxes.

The semantics of OWL is also based on $\mathcal{SROIQ}$. This has the additional advantage that OWL can make use of a wide range of existing reasoners that have been developed for performing reasoning tasks on $\mathcal{SROIQ}$-based knowledge bases.

2.4.6. Unique Name and World Assumptions

OWL and description logics do not apply the UNA, i.e., they do not assume unique names for individuals. This means that individuals that have different names are not assumed to be different. It must be explicitly stated that two individuals are different using the *DifferentIndividual(*$a_1 ... a_2$*)* assertion (i.e., $a_1 \neq ... \neq a_2$ in description logics).

Another important assumption is whether the set of individuals is considered complete or not. This is called the *world assumption* [KSH12]. One distinguishes between the *Open World Assumption (OWA)* and the *Closed World Assumption (CWA).* Description logics and OWL both apply the OWA. Open world describes knowledge in an extensible way by assuming that the knowledge stated in the ABox is incomplete. The absence of information in the knowledge base simply indicates a lack of knowledge. This is due to the fact that knowledge representation systems are mostly applied where it cannot be assumed that the knowledge given

Table 2.4.: *An excerpt of OWL expressions and their mapping to description logics where C and D are class expressions, P is a property expression, and a, b are individuals. The prefix owl refers to the namespace of the OWL definition.*

	OWL Functional-Style Syntax	**Description Logics Syntax**
Class Expression Axioms	*SubClassOf(C D)*	$C \sqsubseteq D$
	EquivalentClasses(C ... D)	$C \equiv ... \equiv D$
	DisjointClasses(C ... D)	$C \sqcap ... \sqcap D \sqsubseteq \bot$
	DisjointUnion(C C_1 C_2 ... C_n)	$C_1 \sqcup C_2 \sqcup ... \sqcup C_n \equiv C$ and $C_1 \sqcap C_2 \sqcap ... \sqcap C_n \sqsubseteq \bot$
Object Property Axioms	*InverseObjectProperties(P_1 P_2)*	$P_1 \equiv P_2^-$
	SubObjectPropertyOf(P_1 P_2)	$P_1 \sqsubseteq P_2$
	ObjectPropertyDomain(P C)	$\exists P.\top \sqsubseteq C$
	ObjectPropertyRange(P C)	$\top \sqsubseteq \forall P.C$
	FunctionalObjectProperty(P)	$\top \sqsubseteq 1P$
Assertions	*ClassAssertion(C a)*	$C(a)$
	SameIndividual(a_1 ... a_n)	$a_1 \approx ... \approx a_2$
	DifferentIndividual(a_1 ... a_n)	$a_1 \neq ... \neq a_2$
	ObjectPropertyAssertion(P a b)	$P(a,b)$
Class Expressions	*ObjectIntersectionOf(C D)*	$C \sqcap D$
	ObjectUnionOf(C D)	$C \sqcup D$
	ObjectComplementOf(C)	$\neg C$
	owl:Thing	$\top$
	owl:Nothing	$\bot$
	ObjectSomeValuesFrom(P C)	$\exists P.C$
	ObjectAllValuesFrom(P C)	$\forall P.C$
	ObjectMinCardinality(n P C)	$\geq nP.C$
	ObjectMaxCardinality(n P C)	$\leq nP.C$
	ObjectExactCardinality(n P C)	$= nP.C$

in the knowledge base $\mathcal{KB}$ is complete. The OWA allows for underspecification. This enables reuse and extension of the knowledge base, since OWL and description logics are monotonic, i. e. existing statements remain true when adding information to the knowledge base.

Modeling languages (e.g., UML) and databases assume that the set of individuals of a given model is complete, i.e., that no information is missing. They assume that there is only one interpretation, i.e., they apply a CWA. Everything that does not belong to this interpretation belongs to its complement. In a CWA, class axioms are considered as integrity constraints [Sir10]. Missing assertions are able to create integrity violations. For example, the axiom $Repository \sqsubseteq \exists manage.Entity$ ("Each repository must manage an entity") is violated if the assertion $manage(repo1, entity1)$ is missing for existing entities $Repository(repo1)$ and $Entity(entity1)$. In contrast to this, those exemplary entities would not create a violation when an open world is assumed, since it is assumed that the assertion $manage(repo1, entity1)$ is not specified yet in the ABox. Existing reasoner implementations such as Pellet [SPG$^+$07] support open world reasoning by default. However, some reasoners also allow for reasoning on a knowledge base using closed domains [SPG$^+$07].

2.4.7. The Semantic Web Rule Language

The expressiveness of OWL can be extended by adding so-called *rules* to the language. The Semantic Web Rule Language (SWRL) [swr04] is a standardized language for expressing such rules. It provides a high-level abstract syntax for expressing Horn-like rules [Hor51]. With such rules, knowledge can be described and defined. Rules can be used to infer new knowledge. An SWRL rule is written as follows:

$$b_1, b_2, ..., b_n \rightarrow a$$

SWRL rules consist of an implication between an antecedent, i.e., the body $(b_1, b_2, ..., b_\mathrm{n})$, and a consequent, i.e., the head (a). Intuitively, such a rule can be read as *"whenever the conditions specified in the body are satisfied, then the conditions specified in the head must also hold"*. An empty antecedent implies true, while an empty consequent is interpreted as false. The body and the head consist of *atoms*. Atoms can be of the form $C(x)$ or $P(x,y)$, where C is a concept and P is a role. x and y are variables, individuals or data values. Predicates with one parameter of the form $C(x)$ are concept assertions. C is the name of an OWL class and x is an individual name. The predicate holds iff x is an instance of C. $P(x,y)$ holds iff x has the property y, i.e., the individual is related with the individual y via the object property P or is related with the value y with the data property P.

The extension of OWL with rules is undecidable. In order to preserve decidability, rules are restricted to so-called DL-safe rules [Krö10a]. DL-safety ensures that each variable is bound only to individuals explicitly introduced in the ABox. For example, the rule

$$parent(x,y), brother(y,z) \rightarrow uncle(x,z)$$

can be extended as follows

$$O(x), O(y), O(z), parent(x,y), brother(y,z) \rightarrow uncle(x,z)$$

where O is not a concept from the knowledge base. This ensures that every variable appears in the rule condition and that reasoning stays decidable. In Chapter 7, it is shown how SWRL rules can be used to infer architectural information from source code, i.e., how to extract the implemented architecture.

2.4.8. SPARQL

SPARQL [W3Cc] is the triple-based W3C standard query language for RDF graphs. Its syntax is syntactically similar to the SQL language for querying RDF graphs via graph pattern matching. Basically, queries follow a `SELECT-FROM-WHERE` structure as shown in Figure 2.6 describing a graph pattern.

Listing 2.1 shows a simple, exemplary SPARQL query without a `FROM` clause. It uses the example RDF graph shown in Figure 2.5 in order to retrieve the title of a book and the author who has written the book.

Listing 2.1: *Exemplary SPARQL query for the data set illustrated in Figure 2.6.*

```
PREFIX ex: <http://example.org/>

SELECT ?title ?author
WHERE
 {
   ?book ex:hasName ?title .
   ?author ex:isAuthorOf ?book
 }
```

Three major parts of a query can be seen here: `PREFIX`, `SELECT`, and `WHERE`. The `PREFIX` defines prefixes as abbreviations of namespaces defining the URI of the corresponding RDF data sets. The `SELECT` clause defines the query result. It introduces variables (here: `?title` and `?author`) that are used to define the parts of a query pattern that should be retrieved as a result. The actual query is introduced by the `WHERE` clause. The query is defined by listing triples separated by a full stop. In the example in Listing 2.1, two triples are listed. The two triples define the graph pattern that should be matched. This pattern describes the RDF graph that should be searched for in a given data set. When the pattern can be matched, concrete values are bound to the variables. Assuming the data set in Figure 2.5, the result would be as follows:

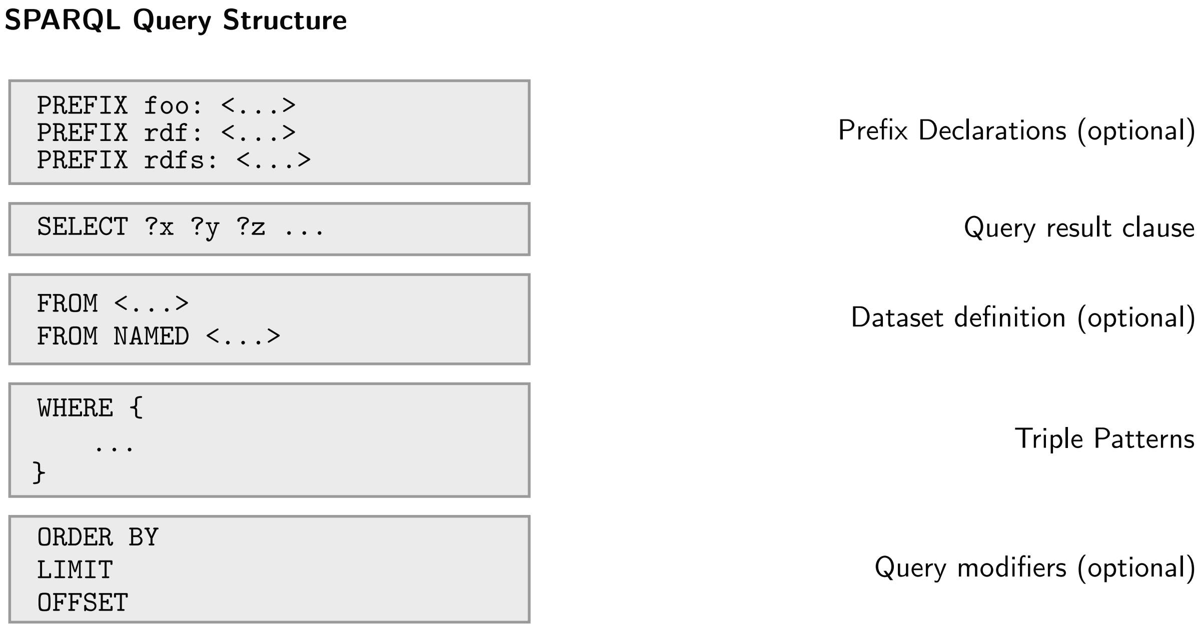
`title`	`author`
“Semantic Web for Beginners”	http://example.org/johndoe

In an optional `FROM` or `FROM NAMED` clause, a specific data set the graph pattern should be applied on, can be explicitly selected. The last optional part of a SPARQL query are query modifiers. Query modifiers allow for controlling how a result set should be returned, e.g., limiting the number of results that should be returned, which is a supportive operator in case of large result sets.

This section only aims to provide the basics of SPARQL queries so that the structure of queries as used in Section F.1 can be understood. For a more comprehensive introduction into SPARQL, the reader is referred to [HKR09].

2.5. Controlled Natural Languages

Natural languages are the most expressive languages that are easy to use and to understand by humans. However, natural languages are ambiguous, leaving room for interpretations. Therefore, they are difficult to be processed and understood by computers. CNLs aim for


SPARQL Query Structure
Prefix Declarations (optional)
Query result clause
Dataset definition (optional)
Triple Patterns
Query modifiers (optional)

```
PREFIX foo: <...>
PREFIX rdf: <...>
PREFIX rdfs: <...>

SELECT ?x ?y ?z ...

FROM <...>
FROM NAMED <...>

WHERE {
    ...
}

ORDER BY
LIMIT
OFFSET
```

Figure 2.6.: *General structure of a SPARQL query.*

reducing the ambiguousness and the complexity of natural languages. CNLs are a subset of natural languages that are restricted in terms of their grammar and vocabulary. Kuhn et al. define CNLs as follows:

Definition 2.5.1: Controlled Natural Language [Kuh14]

A Controlled Natural Language (CNL) is a constructed language that is based on a certain natural language, being more restrictive concerning lexicon, syntax, and/or semantics, while preserving most of its natural properties.

The definition by Kuhn et al. emphasizes that a CNL is *constructed*, which means that the language has an explicit definition and is not the result of an implicit and natural process [Kuh14].

CNLs are designed for different purposes. Basically, two types of CNLs can be identified based on their purpose [Sch02], namely CNLs that aim for 1) improving communication among humans and 2) providing a natural and intuitive representation for formal notations. In this thesis, the approach strives for both purposes (see Section 1.2).

Intuitively, CNLs are considered more formal than natural languages and are therefore able to be understood by computers. Additionally, they are more natural than formal languages. In this way, they can act as a natural language interface that allows humans to unambiguously write sentences while those sentences can be processed by computers due to their formality. CNLs are something in between natural languages and formal languages. A natural language is very expressive, but complex and imprecise, while a formal language is very precise, but at the same time unnatural. Roughly speaking, CNLs are a compromise between natural and formal languages.

2.5.1. Existing CNLs

Kuhn differentiates between General-Purpose CNLs, CNLs for Business Rules, and CNLs for the Semantic Web [Kuh10]. In the following, some existing CNLs from each category are shortly described.

General-Purpose CNLs: Attempto Controlled English (ACE) [Kuh10], Processable English (PENG) [WS09], and Common Logic Controlled English [Sow] are general-purpose languages. All three CNLs have English as their base language. Sentences written in ACE can be automatically and unambiguously translated into first-order logic. PENG covers a smaller subset of natural language than ACE and therefore constitutes a more lightweight approach. Those languages are considered general-purpose CNLs, since they are not designed for a specific application domain.

CNLs for Business Rules: Having business rules formalized in an appropriate format has several advantages. First, business rules are clearly defined for all stakeholders, and, secondly, business rules can be automatically processed by business rule systems. Business rules need to be approved and read by humans who normally do not have a background on logical languages. That is why, an intuitive representation in form of a CNL can be supportive. RuleSpeak [Sol], RuleCNL [NCGA14], and SBVR Structured English [LN13] are examples of such languages. Specification languages like Cucumber [cuc] can also be considered as CNLs. These languages are used in the context of behavior-driven development to describe expected software behavior that can be understood by customers that are normally no experts in software engineering.

CNLs for Semantic Web: In the previous section, OWL and description logics have been presented as a means for describing ontologies. For OWL, several syntax variations have been developed. However, not all ontology representations are intuitive for users [Kuh13]. That is why, attempts were made to make ontology design more intuitive. CNLs like Rabbit [DDC$^+$10], CLOnE [FTB$^+$07], and Lite Natural Language [BCT07] provide natural language interfaces for ontology authoring. Sentences written with these languages are directly translated into OWL. Sentence patterns can be mapped to OWL axiom patterns.

The CNLs of interest in the context of this thesis are those used for ontology authoring (CNLs for Semantic Web). *ArchCNL* adopts several grammatical elements of existing CNLs as will be described in Chapter 6.

3. An Empirical Study on Architecture Enforcement Concerns and Activities in Practice

In literature, a lot of approaches are concerned with developing improved methods for supporting architecture enforcement. However, less work is concerned about the actual perspectives and objectives the architects have in practice. In this chapter, results of an empirical study performed with experienced software architects are presented. Using grounded theory as a qualitative analysis method, architecture enforcement concerns and activities are revealed. The results of the study contribute to the characterization of the architecture enforcement process and therefore contribute to goal **G1** as described in Chapter 1. Parts of this chapter have been published in **[Sch16]** and **[Sch18b]**.

3.1. Goal of the Study and Research Questions

A lot of approaches to monitor and control architecture erosion have been proposed, e.g. [TV09], [PKvdWB14] and [HMRS13]. However, current approaches do not empirically investigate architecture enforcement in practice, and especially how architects face problems for achieving architecture enforcement. This is important to ensure the that novel approaches and tools integrate well into the enforcement practices of the software architect. Therefore, one goal of this thesis is to understand architecture enforcement in practice from the software architects' perspective. The second goal of the thesis is to determine and develop methods to support architecture enforcement. In detail, the following research questions are investigated in this thesis:

- **RQ1: What are the concerns architects consider during architecture enforcement?**

With this question, concrete objectives (e.g., adherence to architecture design principals or properties) to achieve the two main goals of architecture enforcement (see Chapter 1) are determinated. Those objectives are called *enforcement concerns.* By capturing, prioritizing and categorizing enforcement concerns from real examples in practice, empirically grounded requirements for new approaches and tools to support architecture enforcement are specified. This can provide directions for further research on supporting architecture enforcement.

- **RQ2: What are the activities performed by architects during architecture enforcement?**

The motivation of RQ2 is to determine current activities that an architect performs during architecture enforcement. Those activities are called *enforcement activities.* By determining

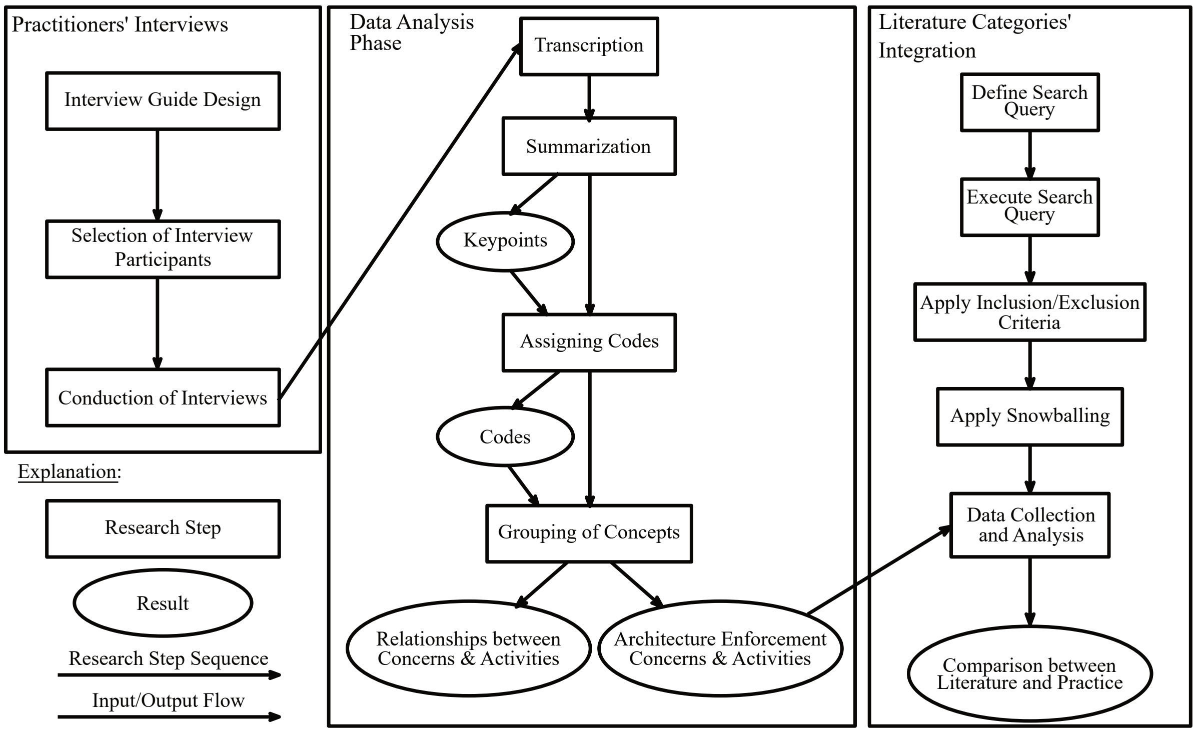

Practitioners' Interviews
Interview Guide Design
Selection of Interview Participants
Conduction of Interviews
Explanation:
Research Step
Result
Research Step Sequence
Input/Output Flow
Data Analysis Phase
Transcription
Summarization
Keypoints
Assigning Codes
Codes
Grouping of Concepts
Relationships between Concerns & Activities
Architecture Enforcement Concerns & Activities
Literature Categories' Integration
Define Search Query
Execute Search Query
Apply Inclusion/Exclusion Criteria
Apply Snowballing
Data Collection and Analysis
Comparison between Literature and Practice

Figure 3.1.: *Overview about the overall study design.*

and understanding current enforcement activities, types of scenarios are determined that are relevant in practice. Taking this into consideration will increase the probability of acceptance and usefulness for new approaches and tools.

- **RQ3: How do architecture enforcement activities support fulfilling architecture enforcement concerns?**

The motivation behind RQ3 is to determine the reason behind conducting each of the enforcement activities. In this way, architecture enforcement activities and concerns are related. Concrete relationships between architecture enforcement activities and concerns determine concrete scenarios for architecture enforcement.
In order to answer those research questions, an empirical study with 17 experienced software architects was conducted.

3.2. Research Process and Study Design

The study applies a qualitative research approach. The process comprises two main phases: *Practitioners Interviews* and *Literature Categories' Integration.* The first phase aims to discover the current state of the practice of architecture enforcement. The findings of the first phase are then connected with findings from the second phase which collects relevant publications related with architecture enforcement. For the first phase, expert interviews based on a semi-structured interview guide are performed. The interviews were transcribed and analyzed by adopting coding procedures from grounded theory, e.g. as described by [SC$^+$90]. In the second phase, a comparison of our findings with those found in literature was conducted. Steps as performed in a systematic literature review [KC07] are conducted to find literature that could be relevant to architecture enforcement. The design and the corresponding results of the literature review are not shown here, but presented in the appendix in Chapter B. The concepts and categories derived from the interview study are used in order to label phrases in the publications found during the literature collection process. With this, it is examined to which extent the findings from the interviews are covered in current literature. Figure 3.1 gives an overview on the overall research process. It depicts the sequence of research steps (boxes) performed in the study and the corresponding results of the research steps (ellipses). In the next section, the design of the expert study and the interview analysis are explained in more detail.

3.2.1. Data Collection

The current state-of-the-practice of architecture enforcement is investigated by conducting qualitative research. For this, an interview study with semi-structured interviews is designed by following the guidelines of [Cha14]. Those interviews are an integral part of qualitative research. They aim to generate new knowledge about a specific topic about which only a few findings exist [HA05] and support to collect as much new knowledge as possible. An interview guide supports keeping the focus on our research questions during the interview. Moreover, the guide ensures that the same set of questions was asked in all interviews. The interview guide was designed for a semi-structured interview containing open questions that were chosen according to the research questions. Those questions allow the participants to talk freely about their experience concerning architecture enforcement. The interview guide contains three parts. The first part classifies the experiences and the technical background of each participant, such as the application domain, years of experience, the development process, the team size, experiences regarding a specific technology etc. The second part contains open questions that aim for identifying architecture enforcement concerns. Correspondingly, the third part contains questions addressing the enforcement activities that software architects perform. The participants are asked which concrete methods they apply in order to enforce architecture. The interview guide was tested with a master student and a researcher before interviewing the selected participants. Table 3.1 depicts the participants that have been selected for the interviews. The master student and the researcher are not included in this table. The first 12 interviews (**A**-**L** in Table 3.1) were conducted by the author of this thesis. Another researcher conducted the other five interviews (**M**-**Q** in Table 3.1). The same interview questions were used. Additionally, the concepts identified from the first 12 interviews were presented to the latter 5 participants in order to examine their perspectives and opinions on those concepts. The detailed interview guide is given in the appendix (see Section A.1). Experienced software architects from industry are chosen as participants of the study. Architects come from different companies from Germany, Switzerland, and the USA. All study participants hold at least a master's degree or a similar qualification in computer science or related fields, e.g. electrical engineering or physics. In total, 17 architects from 16 different companies were interviewed. The interview participants are listed in Table 3.1. The professional experience of the participants ranged from 5 to over 20 years, with an average of 13 years. All of them work as a software architect or have significant practical experience in architecture design. In the first interview phase, software architects that solely work in the enterprise domain (such as banking, logistic etc.) were interviewed. In the second interview phase, two more software architects from the enterprise application domain were consulted in order to validate the concepts. Additionally, it is investigated whether new concepts emerge with the additional interviews. As no new

Table 3.1.: *List of study participants, their domain, their years of experience, and the size of the team they are supervising.*

#	**Domain**	**Role(s)**	**Exp. (years)**	**Team size (approx.)**
A	enterprise (application, integration)	software architect	> 15	100
B	enterprise	software architect, consulting	10 – 15	10 – 20
C	enterprise	software architect	> 20	10
D	logistic	software architect, agile test engineer	10	5-10 per component team
E	accounting / enterprise (migration)	software architect, section manager	10 – 15	50
F	enterprise	software architect, lead developer	10 – 15	2 – 7
G	enterprise / embedded	software architect, coach	10 – 15	10
H	insurance	software architect, project manager	5 – 10	10 developers, 10 test engineers
I	medical	software architect, software developer	5 – 10	10
J	government / enterprise (application)	software architect, consulting	10	10
K	logistic / enterprise	software architect	5 – 10	5 – 10 per component team
L	banking, control systems, enterprise	software architect, project manager	> 20	10
M	enterprise	technical leader	10 – 15	10 – 20
N	retail and health-care enterprise	software architect, project manager	> 20	30
O	Automotive	project manager	10 – 15	10 – 20
P	Automotive	software architect	10 – 15	50
Q	Automotive	project manager	10 – 15	10

concepts emerge, software architects from the automotive domain are interviewed additionally. In this way, it is investigated whether software architects from the automotive domain consider other concerns or perform other activities and how they prioritize the concerns and activities that were already discovered.

3.2.2. Data Coding and Analysis

The interviews were transcribed word-by-word and then analyzed qualitatively by conducting open coding [SC+90]. Instead of defining codes before analyzing the interviews, the categories directly emerge from the data. The only restriction was that the process explicitly focuses on searching for data that may indicate enforcement concerns and activities. However, no coding scheme was defined beforehand in order to be open minded during the coding process. The results of the coding process can be found in the appendix in Chapter A. For the coding process, *AtlasTi* [Atl] is used in order to support the codification process. The following analysis steps are performed:

1) Summarization: In a first step, the main point of the raw data is summarized in a few words in order to generalize the raw data. This can be only a few words or a whole phrase if necessary.

2) Assigning codes: The main point is then assigned to a code. [HM16] used a similar technique where they first summarize the raw data into so-called key points and then assigned a code to them.

3) Grouping of concepts: The codes assigned to the data are continuously compared with the codes assigned in other interview transcriptions. This is called the constant comparison method [Gla78]. Constant comparison helps to achieve a higher level of abstraction by finding concepts. By iteratively applying the constant comparison method on the emerging concepts, categories are identified. Each category encompasses a group of concepts that appear to relate to the same phenomenon.

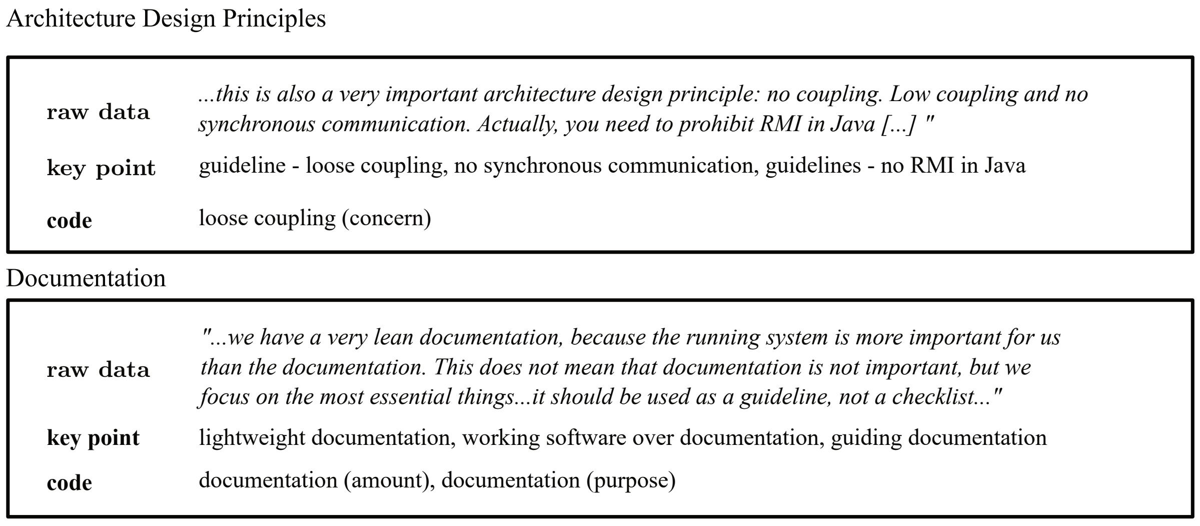
The excerpt in Figure 3.2 shows two examples for the categories "Architecture Design Principles" and "Documentation". The corresponding interview data, key points, and codes are also depicted. As can be seen, the data related to the concept "Architecture Design Principles" is labeled with the code "loose coupling". Additionally, other phrases in the interviews could be identified that were assigned to the concepts separation of concerns, dependencies, modularization and so on. Those concepts were finally grouped into the category "architecture design principles" that is classified as an enforcement concern. The brackets after a code express properties, dimensions, or other characteristics related with it. This is depicted in the example for the concept "Documentation". Instead of simply assigning the code "documentation" to this excerpt, the properties in the brackets are used in order to give a lead that this data is about the amount and the purpose of the documentation in this context.

3.2.3. Axial Coding

Additionally, axial coding [Cha14] is applied in order to find connection between codes. This involves documenting category properties and dimensions from the open coding process by identifying conditions, actions, and interactions with a specific phenomenon and relating categories to subcategories. Axial coding involves the following components:

Causal Condition: conditions that influence the core phenomenon,

Core Category/Phenomenon: the central idea or incident about which a set of actions or interactions is directed,



Architecture Design Principles

raw data	*...this is also a very important architecture design principle: no coupling. Low coupling and no synchronous communication. Actually, you need to prohibit RMI in Java [...] "*
key point	guideline - loose coupling, no synchronous communication, guidelines - no RMI in Java
code	loose coupling (concern)

Documentation

raw data	*"...we have a very lean documentation, because the running system is more important for us than the documentation. This does not mean that documentation is not important, but we focus on the most essential things...it should be used as a guideline, not a checklist..."*
key point	lightweight documentation, working software over documentation, guiding documentation
code	documentation (amount), documentation (purpose)

Figure 3.2.: *Data coding example for the concepts "Architecture Design Principles" and "Documentation".*

Strategies: actions or interactions addressing the phenomenon,

Context: context in which the phenomenon and corresponding strategies apply,

Intervening conditions: conditions that shape, facilitate or constrain the strategies,

Consequences: outcomes or result of the strategies.

Using this coding scheme, relationships between enforcement concerns and activities are revealed, so that it is explained which activity is preferably conducted by the architect for a specific concern. Figure 3.3 shows an example of axial coding that is explained in the following. During enforcement, software architects aim to ensure the agreement on design decisions. This is the core category that is considered during this axial coding example. Several activities can be conducted by the software architect in order to reach this goal. However, the type of activity is influenced by some causal conditions. In this case, those are the programming habits, skills, and experiences in the development team. When enforcing architecture patterns (context), the software architect may adjust the software architecture according to the developers' skill (strategy). This activity is constrained by two intervening conditions: 1) the software architect needs to choose a pattern with which developers are familiar and 2) the pattern still needs to be appropriate for the given functional and non-functional requirements. Applying this activity successfully results in a feasible architecture that can be implemented by the developers and that is accepted by them (consequences).

3.3. Enforcement Concerns

In this section, the enforcement concerns that are discovered from the transcribed interviews are presented. The concerns can be divided into two categories, namely *Design Decision* and *Implementation Quality* as can be seen in Figure 3.4. The category *Design Decisions* corresponds to the most important decisions that architects consider during architecture enforcement. The category *Implementation Quality* encompasses all those concerns that an architect has regarding

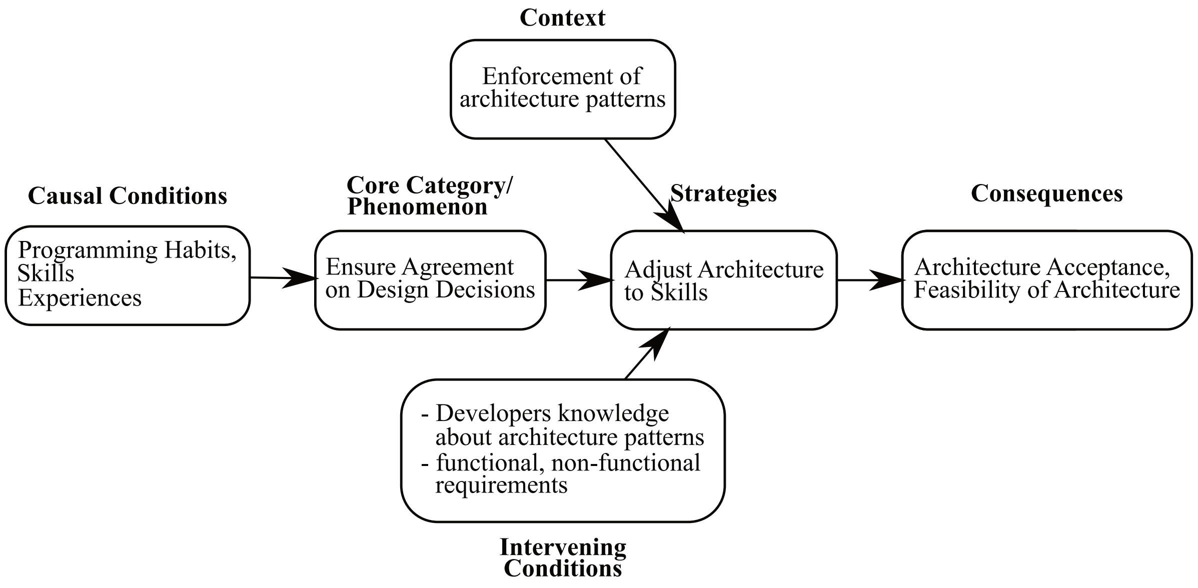

Context
Enforcement of architecture patterns
Causal Conditions
Programming Habits, Skills Experiences
Core Category/ Phenomenon
Ensure Agreement on Design Decisions
Strategies
Adjust Architecture to Skills
Consequences
Architecture Acceptance, Feasibility of Architecture
- Developers knowledge about architecture patterns
- functional, non-functional requirements
Intervening Conditions

Figure 3.3.: *Axial coding example.*

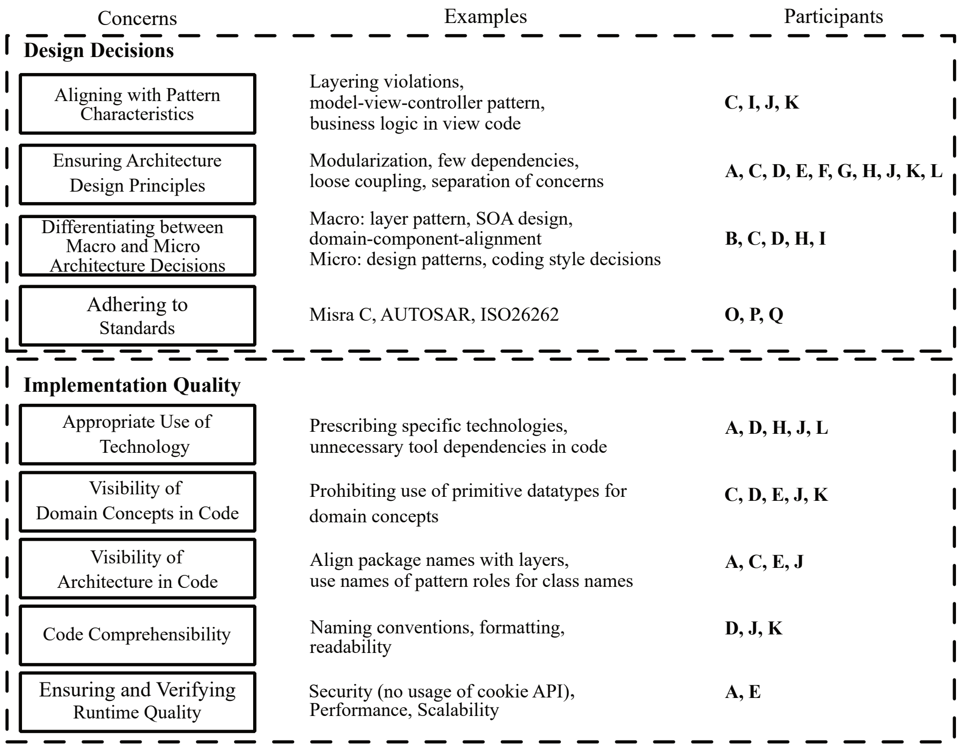
the correct implementation of those architecture design decisions. Figure 3.4 shows how the investigated concerns are mapped to their respective category. The concerns will be explained in more detail in the following sections. Each concern is supported with quotes from the interviews. Additionally, it is shown which concern is mentioned by whom. Please note that the interviews have been conducted in German. That is why, the quotes have been translated to English.

3.3.1. Aligning with Pattern Characteristics

Participants mentioned that they aim to ensure that important pattern rules are followed in the implementation. They explicitly differentiate between architecture patterns on a higher level and design patterns on code level. As emphasized by the participants, design patterns are actually not checked by them, except if they are crucial for specific non-functional requirements. That is why, they sometimes validate which design patterns are implemented and if they fit in the specific context: *"which design patterns are used and in which context. Are they only used just because I have seen it in a book or because I wanted to try it or is it really reasonable at this place..."* (code: pattern suitability, Participant **C**). The Layer and the Model-View-Controller pattern [Fow02] where mentioned regularly in the interviews as important patterns that need to be controlled during implementation. Regarding the layer pattern, dependency violations often occur and need to be controlled: *..."when we decided for a layer architecture we took care that the layering is ensured..."* (Participant **G**) or *..."that I only have the defined relations between the layers"* (Participant **J**).

3.3.2. Ensuring Architecture Design Principles

Software architects especially focus on architecture design principles, such as modularization, separation of concerns, or loose coupling: *"[the system] is composed of very loosely coupled modules that only communicate asynchronously [...] this is also a very important architecture principle: loose coupling..."* (codes: loose coupling, modularization; Participant **L**). Experts



Concerns	Examples	Participants
Design Decisions		
Aligning with Pattern Characteristics	Layering violations, model-view-controller pattern, business logic in view code	**C, I, J, K**
Ensuring Architecture Design Principles	Modularization, few dependencies, loose coupling, separation of concerns	**A, C, D, E, F, G, H, J, K, L**
Differentiating between Macro and Micro Architecture Decisions	Macro: layer pattern, SOA design, domain-component-alignment Micro: design patterns, coding style decisions	**B, C, D, H, I**
Adhering to Standards	Misra C, AUTOSAR, ISO26262	**O, P, Q**
Implementation Quality		
Appropriate Use of Technology	Prescribing specific technologies, unnecessary tool dependencies in code	**A, D, H, J, L**
Visibility of Domain Concepts in Code	Prohibiting use of primitive datatypes for domain concepts	**C, D, E, J, K**
Visibility of Architecture in Code	Align package names with layers, use names of pattern roles for class names	**A, C, E, J**
Code Comprehensibility	Naming conventions, formatting, readability	**D, J, K**
Ensuring and Verifying Runtime Quality	Security (no usage of cookie API), Performance, Scalability	**A, E**

Figure 3.4.: *Discovered enforcement concerns, examples mentioned in interviews and corresponding participants who mentioned the concern.*

also aim to control the dependencies between components, so complex structures and high coupling are minimized or even avoided. Participant **H** especially stated that developers tend to create monolithic structures that are hard to maintain and deploy.

3.3.3. Differentiating between Macro and Micro Architecture Decisions

The interviewed participants differentiate between two abstraction levels on which architecture decisions can be made. These are so-called macro architecture and micro architecture decisions. Some participants also used other terms like strategic or global (i.e., macro) and tactical or local (i.e., micro). This distinction is used in order keep the focus on the most important decisions – that are related with the macro level – and to delegate decisions with local impact to developers. The participants define the macro architecture as the general idea or or metaphor of a system [CB11] with its most critical architecture decisions. This may encompass the fundamental architectural style, structures, data stores, or communication style: *"...it is important how you regard it. For me there are basically two views about how software is built. First you have the global view [...] There I decide how I design my software, for example using Domain Oriented Design or SOA."* (code: two different views of architecture, Participant **D**) or *"...then we have the micro architecture, this is the architecture within each team. A team can decide for its own component for which it is responsible which libraries it wants to use."* (codes: two different views of architecture, macro architecture, micro architecture; Participant **K**). The interviewed participants report that they basically focus on the macro level of software architecture and consider the micro level as developers' responsibility.

3.3.4. Adhering to Standards

In some domains, which involve high risks regarding criticality and safety, the adherence to standards guarantees the achievement of quality requirements and mitigate risks. For example, software development in the automotive domain uses standards for software architecture design (e.g., AUTOSAR [AUT]). Some standards additionally specify levels, where each level adds rules for the design and implementation of software. For example, the ISO 26262 [ISO11] has an Automotive Safety Integrity Level, which specifies the degree of safety. Software architects need to select suitable standards and levels, and make sure that developers adhere to the rules defined by standards during their implementation. One of the interview participants mentioned *"Safety standards are important (...) Not adhering to standards can threaten the life of people".* Moreover, architects are concerned with adherence to code quality standards (e.g., Misra C [MIS]). This makes it easier for architects to guarantee good code quality with less effort, because code quality standards could be checked using tools.

3.3.5. Appropriate Use of Technology

Some participants mentioned that developers often tend to use a lot of tools and technologies that are not necessary: *"...aim for technologies is the biggest problem. And if you like to use those frameworks because they are providing advanced functionalities, but you cannot control those functionalities if you do not have enough experiences with it..."* (code: aim for technologies, Participant **J**). Due to their complex functionalities and numerous ways to be used, it is necessary that software architects also control the way how developers apply a specific technology, since inappropriate use can potentially violate the conceptual integrity and consequently can cause architecture erosion.

3.3.6. Visibility of Domain Concepts in Code

Domain concepts relevant for a specific application domain should be visible in source code elements as mentioned by the participants. That is why, they aim to use terms for code elements derived from concepts from the application domain: *"...I like to be guided by the domain instead of using technical terms [...] both can work, but from my experience using domain oriented terms is easier to understand..."* (code: domain oriented terminology, Participant **J**). This has several advantages during the development. Firstly, it helps to talk with domain experts about the software design. Secondly, architects and developers can easily locate the relevant code locations that needs to be changed in case that requirements evolve. Another architect emphasized that it *"should be clear which part of the source implements which functionality"* (Participant **D**).

3.3.7. Visibility of Architecture in Code

Similar to domain concepts, concepts related to software architecture should also be made explicit in the code, e.g. by naming code elements after pattern concepts: *"...therefore it is important that the architecture is recognizable in source code. This is absolutely essential for the structure of the project."* (code: making architecture visible in the code, Participant **J**). It additionally helps the architect to locate architecture decisions in the code in order to validate their corresponding implementation.

3.3.8. Code Comprehensibility

Inconsistent use of naming conventions and coding styles decreases code comprehensibility. Participants mentioned that incomprehensible code significantly contributes to architecture erosion. If the code cannot be understood by other developers this may lead more likely to architecture violations: *“if you strictly follow this approach then you have very readable code. From my experience, readable code tends to be more stable. This means, it is easier to implement code that is conform to the architecture and does not have any [architecture] violations...”* (codes: code comprehensibility, code comprehensibility supports architecture conformance; Participant **J**).

3.3.9. Ensuring and Verifying Runtime Quality

Interviewed participants mentioned that they are concerned with ASRs related with security, performance, and scalability. That is why, they aim to ensure that there are no ASR-violating code statements in the implementation: *“then you investigate the code and validate if it fulfills the ASRs. [...] if elements from the domain model, such as orders or credit card information, are stored in a cookie, then this is a violation obviously regarding the decision "Data-based or Server Session State". This has highest the priority and needs to be repaired immediately.”* (codes: ASR violating code structures, practice – checking ASRs; Participant **A**).

3.4. Enforcement Activities

In the following, the results of the second research question are presented. For this, the main open-ended question *“How do you ensure that your architecture and your concerns are implemented as intended? Do you follow any strategies?”* is asked. The result is a collection of enforcement activities as depicted in Figure 3.5. The activities are categorized according to the enforcement goals from Section 2.3 *Ensuring the agreement on design decisions with stakeholders* (abbreviated as *Ensure agreement*) and *Check the conformance between implementation and design decisions* (abbreviated as *Conformance Checking*). In the following, each enforcement activity category and the corresponding activities are presented and supported by statements from the interviews. Here, the statements also have been translated from German to English.

3.4.1. Achieving Mutual Understanding of the Architecture

This category is defined as the activity that aims to achieve consensus about the concepts of the software architecture in the development team. Often, the software architecture is kept in the mind of the developers and the architects. All of them should have the same understanding and picture about the architecture and its underlying decisions: ...*“a common picture – keyword modeling – is very important here, to have a starting point and to have it started in the same direction”* (codes: common understanding of architecture, using models for comprehension, Participant **B**). If a shared understanding is missing in the team, it is more likely that the architecture is (unintentionally) violated by developers. Developers should also have a common understanding about the prescribed architecture, its rationale and its goals that have to be achieved with it: *“skilled people do automatically know how they implement code that is conform to the architecture, because they know, why it should be like that. Then –*

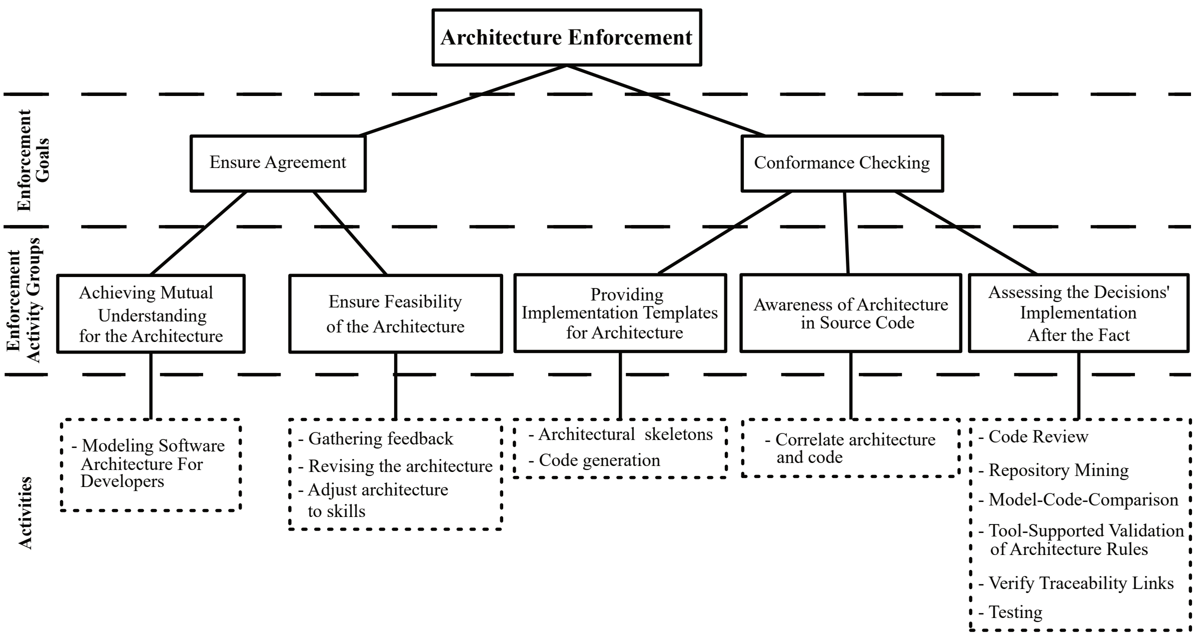

Architecture Enforcement
Enforcement Goals
Enforcement Activity Groups
Activities
Ensure Agreement
Conformance Checking
Achieving Mutual Understanding for the Architecture
Ensure Feasibility of the Architecture
Providing Implementation Templates for Architecture
Awareness of Architecture in Source Code
Assessing the Decisions' Implementation After the Fact
- Modeling Software Architecture For Developers
- Gathering feedback
- Revising the architecture
- Adjust architecture to skills
- Architectural skeletons
- Code generation
- Correlate architecture and code
- Code Review
- Repository Mining
- Model-Code-Comparison
- Tool-Supported Validation of Architecture Rules
- Verify Traceability Links
- Testing

Figure 3.5.: *The mapping of the identified categories of enforcement activities to the respective enforcement goal.*

without help – developers have the architecture in their mind and recognize if architecture goals are fulfilled or not.” (codes: architecture awareness, personal quality; Participant **B**). The risk of introducing architecture violations increases when developers are not aware of architecture goals.

Modeling Software Architecture For Developers

Architecture models greatly help to achieve a common understanding about the architecture. This activity entails two steps: 1) selecting an appropriate modeling notation (formal to informal) for describing the software architecture and 2) the actual modeling of the software architecture. Visualizing software architecture with models is a common activity performed by software architects in order to achieve a mutual understanding and should be considered as an integral part of enforcement. The modeling notation may be formal or informal. For example, participant **B** utilizes diagrams from the UML [RJB04] in order to explain how systems should exchange messages with each other: *“...the most important diagrams are activity diagrams and sequence diagrams. If developers asked us how it should be implemented, we used the diagrams in order to explain [the developers] the specification and make [the developers] understand the planned architecture...”.* Participant **L** uses a more formal modeling language – Fundamental Modeling Concept [KGT05] – in order to support the architecture understanding process. He emphasizes that models are important for adequately communicating and discussing with each other about an architecture design. He argues that it is crucial to have a picture about the planned software system that everybody in the team can understand. This can only be supported by using proper visualizations: *“...the developer needs something that is written down in terms of visual models he can work with. Something that explicitly shows the most important components and processes...”* (code: models and visualizations, Participant **L**). Moreover, he reported that human communication without supportive visualizations embeds ambiguous interpretations.

Therefore he prefers to use models to resolve those ambiguities. Some participants state that informal whiteboard drawings are the most effective and appropriate way to visualize the current architecture, since those drawings are commonly kept in the developers' office and are therefore always present: *"...it is more important that the picture [about the architecture] is kept on a whiteboard in the development team's office, so that the architecture is kept in their sight..."* (codes: whiteboard drawings, shared understanding, documentation(type), Participant **E**).

3.4.2. Ensure Feasibility of the Architecture

The software architect needs to make sure that the developers are able to realize the software architecture by ensuring its feasibility. By doing this, he reduces the risk that the developers might not accept the software architecture as it was designed by the software architect. The architect should always be *"... anxious for getting the architecture accepted by the developers and that they [the developers] want to implement it this way."* (codes: encourage acceptance of developers for architecture, willingness; Participant **B**). In the following sections, three activities are presented that help to ensure the feasibility of architecture.

Gathering Feedback

The interviewed participants aim for regular discussion with the development team, especially in order to address crucial architecture violations or to get feedback from the team regarding the current architecture design. To make this possible, the architect has to be available for feedback as developers might not agree with the architecture solution or the design does not fit the current requirements anymore: *"...we had regular meetings with the developers and showed the developers where deficiencies in the architecture are and where rules were violated."* (code: discussion of violation, Participant **E**).

Revising the Architecture

Participants reported that a software architect needs to be open for potential revisions on the current architecture design. If an architectural solution is too complex to be implemented, the software architect needs to find another solution suitable for the given requirements and simple enough so that it can be implemented by the developers: *"...for example they [the programmers] said that it was not possible to do it differently, because this and that was too complicated, so that we adapted the architecture rules in consequence and said it has to be different here actually, but otherwise it is too complicated."* (codes: solution too complicated, revise architecture, Participant **E**).

Adjust Architecture to Developers' Skills

The software architect should choose architectural solutions (e.g., patterns, technologies) based on the individual skills and experiences in the development team in order to get the solutions accepted. In this way, he can ensure that developers are more familiar with the concepts and to reduce the risk of architecture violations.

3.4.3. Providing Implementation Templates for Software Architecture

Architecture design decisions should be implemented in a uniform way by every developer. The developer's skill and experiences, e.g. from previous projects, and his programming habits influence to a great extent how he implements architectural design decisions. Due to his habits, he can potentially violate those decisions: *"...leaving it to the developers is not suitable since every developer has a different background and experiences. When I just tell them that they should start with programming, then this leads to chaos..."* (code: programming habits and experience of developers; Participant **L**). That is why, the software architect is responsible to provide guidance in the implementation of the decisions. Several activities are conducted by software architects to address this aspect as it was discovered in the interview transcripts.

Architectural Skeletons and Code Generation

In case that architecture concepts are new to developers and the architecture cannot be appropriately adjusted to the team structure, the architect is responsible for coaching and supporting developers adequately. The coaching phase is conducted until all team members are able to implement an architectural solutions. The architect can provide skeletons (or architectural templates) in order to support the coaching process. Using code skeletons he guides developers how a specific decision has to be implemented or a specific technology has to be used. Architectural skeletons provide a reference for the developers during the implementation: *"...you build an example and present it to the developers..."* (code: architectural templates, Participant **A**). Those templates need to be built precisely and carefully according to architecture design decisions and state-of-the-art best practices, as emphasized by participant **A**. Otherwise developers could unintentionally violate underlying decisions. Another way to provide guidance is to generate those templates via code generation. The skeleton code contains the main structure for a single component and the glue code for connecting the component with another one. Developers are forced to stay within the boundaries of the generated code and are not allowed to break the generated code, otherwise important decisions cannot be guaranteed anymore: *"...and then using xslt transformation a skeleton is generated from the XML that is given to the developer and we generate skeletons for the test team. The developer is enforced to work with this skeleton."* (code: code generation, Participant **H**).

3.4.4. Awareness of Architecture in Code

Developers should be aware of when they change code that implements important architecture design decisions. It is the software architect's responsibility to show the developers what are the most crucial code parts regarding architecture design decisions: *"...we showed the developers where [in the code] the architecture rules are actually violated..."* (Participant **E**).

Correlate Architecture and Code

Software developers should be aware whether they are changing architecture-related code. Architecture-related code may be a code part that is responsible for an architecture pattern or tactic implementation. Those parts are significant as they address important quality attributes. In case that architecture violations occur in this specific code part, those quality attributes cannot be guaranteed anymore. That is why, participants emphasized that it is important to clearly describe where and how architecture and the related concepts are implemented. By doing this, the implementation of architectural solutions can be located easier. For example, participant **A** stated that the layer pattern should be clearly mapped onto the package structure. Additionally, classes participating in an architecture pattern implementation should clearly be named after the pattern role. This activity is strongly related with the concern "Visibility of Architecture in Code" described in Section 3.3. This was also favored by Participant **J** who stated that *"...developers are able to orientate themselves easier in the code..."*.

3.4.5. Assessing the Decisions' Implementation after the fact

During the interviews all participants were asked the following question: *"What are the specific steps you perform when you inspect the source code in order to assess the implementation of the architecture decisions?"*. This results in a list of categories describing common methods and activities for assessing the decisions' implementation.

Code Review

In order to assess the implementation according to architecture design decisions, the interviewed participants mostly rely on manual code reviews. One architect stated that this activity *"is similar to the comprehension process of a developer who is new in the team and tries to understand how the software system works. But developers and architects have each different goals during this process. The developer mainly wants to implement new features, while the architect wants to check architecture conformance"* (Participant **C**). During this activity they use their mental model about the software system as the guideline: *"...a picture about if the components are appropriate, if the modules are implemented according to how it was intended..."* (code: expectation about intended design, Participant **C**). In this process, software architects often ask questions about the observed software systems that entail exploration and navigation, such as who implemented this component and where a specific feature, architecture pattern, design pattern, technology is implemented or used. It is then evaluated informally if an implementation roughly represents this mental model. During this process, code analysis tools can be used as a source of information: *"...what you can do is, you run a code analysis tool and then you are looking at the spots that are interesting..."* (codes: finding hot spots, results from code analysis tools as first impression, Participant **K**). Using analysis tools showed to be quite often used to verify the adherence to standards (e.g., MISRA C).

Repository Mining

Repositories and review systems, such as Gerrit [Ger], provide useful information about which changes are made on the software system. One participant mentioned that he uses a review system in order to assess the implementation step by step by reviewing single commits or pull requests of developers. He especially focuses on architectural issues. Using history information, he can easily investigate what type of changes were conducted on a set of classes and especially who did the change. Moreover, introduced architecture violations can be traced back to their emergence. The steps of the implementation can be reproduced and rationale about specific code-level decisions can be reconstructed. If the architect knows about the individual skills in a team, he can focus source code inspections on changes by developers that have less skills, are inexperienced, or are new to a project: *"...you know basically who works on which parts, this* *means if I know from experience that I have to have a closer look on what he or she has created then it is possible that I have to inspect each class [...] because he or she can create an unusual solution on the most unobtrusive parts"* (code: focused inspection based on individual skills of developer, Participant **C**).

Model-Code-Comparison

Participants were asked to which extent architecture documentation and models are used in assessments. Some participants (**B**, **I**, **J**, **L**) use documented diagrams and models for conformance validation between the implemented software system and the architecture. UML class diagrams, sequence diagrams or component diagrams are commonly used. The participants compare those diagrams with models that are automatically extracted from the underlying implementation. The comparison is, however, performed manually. A possible evaluation scenario includes the validation of the message exchange between components and if it conforms to the prescribed behavior given by the documented UML sequence diagram.

Tool-Supported Validation of Architecture Rules

Participants were additionally asked to which degree they formalize architectural aspects in order to allow a formal validation of a software architecture. It is found that participants rarely formalize architecture rules. Some of them formalize the layer pattern and validate whether implemented layer dependencies adhere to the prescribed ones. For this, they use tools such as Sonargraph: *"...actually, automated validation of the macro architecture is conducted with the Sonargraph tool. This is about checking the defined relationships between layers and slices."* (codes: ensuring macro architecture, automatically validate layer pattern; Participant **J**). Some participants also mentioned that they formalize rules on a lower level of abstractions. For example, thresholds for complexity metrics amongst others are validated automatically with code quality tools like Sonarqube [Son] or Checkstyle [Che]. It is worth mentioning that not all architects relate those low-level rules to architecture, but rather to a good programming style. However, some architects aim to be responsible for both: *"...in the strict sense, this is not really architecture, but I think it is better to manage it together [architecture and coding rules] ... the naming convention was given by us [the architecture team]. This is not really architecture, rather programming guidelines, but I think they belong together."* (codes: restricting complexity metrics, naming conventions; Participant **E**).

Verifying Traceability Links

To verify the implementation of design decisions, software architects ask developers to ensure update-to-date traceability links between software architecture design decisions and their implementation. To achieve this, architects and developers use traceability tools (e.g., Reqtify [Req]) to link architecture significant requirements and their corresponding implementation modules. The tools have the ability to parse several documents, and link sentences from different documents to create a traceability tree. In projects that involve different industrial partners, traceability becomes very important, because possible failures and their consequences could be traced and legally proofed. For example, manufacturing of cars involve many different industry partners, each being responsible for one or more components. In case of a problem at a certain component, problems are traced and the responsible partner is identified.

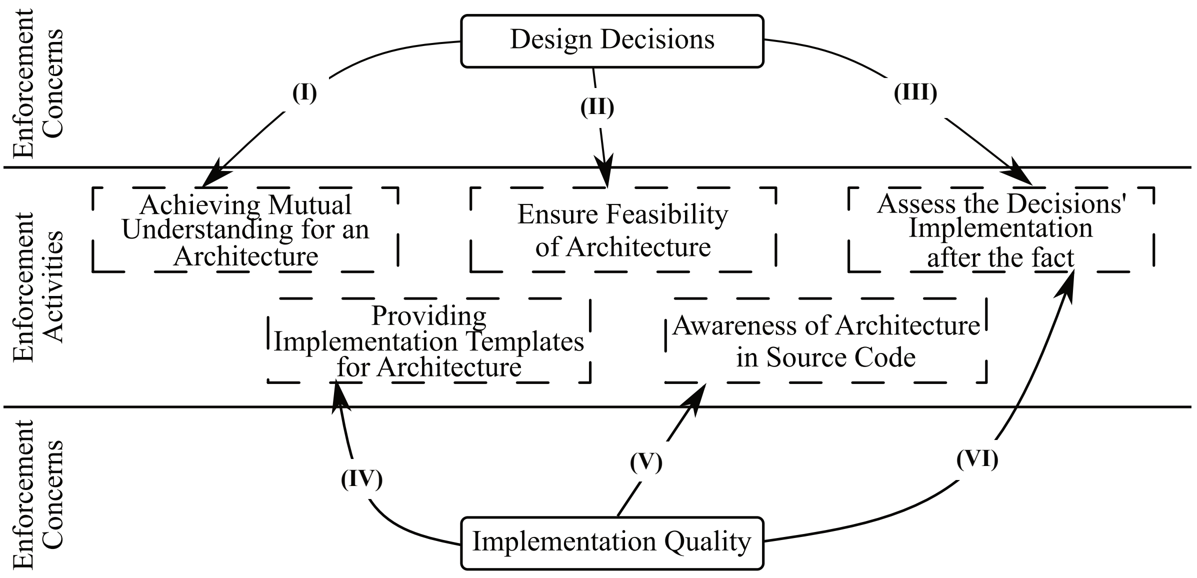

Enforcement Concerns
Design Decisions
(I)
(II)
(III)
Enforcement Activities
Achieving Mutual Understanding for an Architecture
Ensure Feasibility of Architecture
Assess the Decisions' Implementation after the fact
Providing Implementation Templates for Architecture
Awareness of Architecture in Source Code
Enforcement Concerns
(IV)
(V)
(VI)
Implementation Quality

Figure 3.6.: *Relationships between Enforcement Concerns and Activities.*

Testing

Software architects use tests in order to ensure functional and non-functional requirements. They aim to ensure enforcement concerns using appropriate tests. Participants aim for a high test coverage in order to help to discover architecture violations: *"...in case there are only a few tests, then it is likely that people do not build it correctly. This leads to incomprehensible code and consequently to architecture violations."* (code: test coverage supports architecture conformance, Participant **J**).

3.5. Connecting Enforcement Concerns and Activities

In the last two sections enforcement concerns and activities are considered separately. However, connections between those two can be observed in the interview data. This means that software architects prefer specific activities for corresponding concerns. For example, static analysis tools are preferably used in code reviews (the activity) for evaluating code comprehensibility (the concern) by assessing the adherence to naming conventions. Figure 3.6 illustrates those relationships. They are labeled with roman numerals. In the following, the relationships between concerns and activities are described. The discovered relationships are supported with corresponding data from the interviews. For each description, the link is referred to by using the corresponding roman numeral. Again, the two categories of enforcement concerns *Design Decisions* and *Implementation Quality* are distinguished.

3.5.1. Design Decisions

(I) Understanding Design Decisions: Architects apply architecture modeling in order to visualize and explain the main components or patterns to make the developers understand the architecture and increase the shared understanding about the architecture in the development team: *"...models are very important, because humans need something visual, something that can be looked at and that can be understood. Everybody has the same* *picture in their head, because the same picture lies in front of them on the table. [...] We created a big picture about how the server is composed of components and layers..."* (Participant **B**).

(II) Ensuring the feasibility of Design Decisions' Implementation: The software architect needs to ensure that developers are able and are willing to implement an architecture decision by gathering feedback on the decisions, revising them, or by adjusting the decisions according to the developers' skills. For example, the software architect may choose specific patterns that are well-known by the developers in order to increase the architecture acceptance: *"...talking with the developers and examining which reference architectures do they know, do they know the layer architecture or more complex patterns [...] then you take this pattern and try to apply it. This works very well, because the developers feel more comfortable [...] and if you find any violations then they will rather accept them and repair them. Because, they accept this architecture and they want to preserve it. This is much more crucial than having a complex architecture design..."* (Participant **J**).

(III) Validating Design Decisions: As described in Section 3.4, software architects use several methods in order to validate the decisions' implementation. However, they choose a specific method that depends on the type of decision they aim to validate. Architects may use architecture analysis tools like Sotograph [hel] in order to evaluate if the layer pattern is implemented correctly: *"...I validated the layer model with the architecture visualization tool – Sotograph – with this you can perfectly create the layer architecture, in the code we explicitly labeled the layers using a specific naming convention so that it can be directly mapped to Sotograph. The tool showed us the different violations..."* (Participant **E**). However, regarding other macro architecture decisions that cannot be easily checked with tools, like for example the architecture design principle "separation of concerns", architects conduct manual code reviews: *"...what can not be validated with tools or cannot be validated automatically is if a specific functionality belongs to this layer or not or to another layer. This can be only examined through code reviews. And this is something that we have validated through code reviews."* (Participant **E**).

3.5.2. Implementation Quality

(IV) Ensuring quality of decisions' implementation using templates: The concern *appropriate use of technology* (see Section 3.3) is addressed by enforcing a specific way of how a technology or library is to be used. This needs to be enforced in order to address the programming habits of developers who may have different ways of using a specific technology: *"...JMS can be used in six, seven, eight different ways. Can this be decided by the developers? No, it cannot, because everybody uses it differently. That is why we built a framework [...] and we enforced that they only used this framework."* (Participant **L**).

(V) Acknowledging developers about architecture relevant source code: In order to enforce the visibility of architecture in code, another software architect prescribed that the layer pattern was explicitly mapped to the package structure of the code base. Corresponding terms from architecture patterns are used as class names that reify architecture pattern roles. With this enforcement activity, he aims to increase the architecture awareness of the developers during coding: *"...the layer architecture must be clearly recognizable in the* *package structure and you must clearly see where each subsystem is located. [...] projects that are structured in this way helps the developers to orientate themselves in the code easily and to recognize the architecture...”* (Participant **J**).

(VI) Ensuring Code Quality: By performing conformance checks, it can be validated if the implementation quality is ensured in the code according to the concerns described in Section 3.3. Software architects may use static code analysis tools in order to evaluate the overall code quality regarding code comprehensibility (*concern*) by investigating code metrics (*activity*): *“...we use tools that are able to find problems on a code level, like for example big classes with too many methods, method call depths, bad class names and so on [...] this is what is necessary in order to be able to ensure that the code remains clean during changes...”* (Participant **B**). Implementation quality can be enforced by defining specific naming and code structuring conventions, so that the visibility of domain concepts can be ensured. For example, an interviewed participant reported that he strictly prescribes that no primitive datatypes are allowed for domain concepts. In this way, he addressed the programming habits of developers who tend to use only primitive datatypes like int or String to model domain concepts: *“...I started in this project and nobody has ever heard this concept. Never. Everything is modeled as an int, maybe bytes, [...] and you can see that this is not a real type-safe system. [...] they do not build their own data types [...] That is why I enforce that there are no primitive data types at public interfaces [...] they are not thinking about it and simply transform data from one place to another although they are not allowed to. This is indeed a problem.”* (Participant **J**).

3.6. Limitation of the Study

Gasson et al. proposed the criteria confirmability, dependability, internal consistency, and transferability [Gas04] in order to evaluate qualitative studies. By describing and capturing the background of all the study participants transferability is addressed. Confirmability is addressed by repeatedly discussing and restructuring the categories in an iterative process. In order to address dependability a research process has been followed (Section 3.2) and all the steps that were conducted are described. In terms of internal consistency the statements and the corresponding codes were cross-checked by another researcher.

The number of participants in this study is limited. However, since the study aims for generating new knowledge and not to evaluate or confirm existing knowledge, this limited number is quite acceptable.

Another limitation might be that no specific factors that could influence the experts’ view on enforcement concerns are considered. For example, skills and tasks of a software architect could influence his view about what are important concerns and activities in context of architecture enforcement.

There is the risk that each researcher might interpret the results in different ways. This risk is minimized by letting two researchers conduct the interviews independently.
Another limitation may be that the architects were not chosen randomly, but practitioners are directly contacted through the author’s relationships.

3.7. Conclusion

In this chapter, the results of an empirical study investigating the state of the practice of architecture enforcement has been presented. The results of the study provide the motivation of the approach developed in this thesis.

As a main result, the study shows that establishing a common language about the software architecture in order to raise the awareness and to ensure the consensus about the software architecture is of crucial importance (see enforcement activity group *Achieving Mutual Understanding for the Architecture*). A well established language about the software architecture in the project supports the mutual understanding of it. As a consequence, a consistent and well established language can greatly facilitate the architecture enforcement process.

Another essential result of the study is that the architecture must be visible in the source code (see enforcement concern *Visibility of Architecture in Code* and enforcement activity group *Awareness of Architecture in Source Code*). This means that it should be clear how architecture design decisions are implemented so that developers recognize whether their changes have a significant impact on the software architecture. By making the architecture visible in the code, the architecture awareness of software developers can be increased during implementation and consequently, the risk of introducing architecture violations can be reduced.

Last but not least, the results of the study show that it is crucial to verify whether the architecture decisions are implemented as intended (see enforcement activity group *Assessing the Decisions' Implementation After the Fact*). Ideally, architecture decisions should be formalized so that they can be validated automatically to reduce the effort of manual inspection. For this, dedicated tools can be used with which the conformance to architecture decisions can be validated. In the next chapter, existing conformance checking approaches are evaluated with respect to the question whether they are able to support architecture enforcement. Based on the findings of the study, criteria that such approaches must fulfill are defined (Chapter 4). As a next step, a novel ontology-based approach for architecture enforcement is proposed in the subsequent chapter (Chapter 5).

4. State-of-the-Art Analysis of Architecture Conformance Checking Approaches

The results of the empirical study presented in the previous chapter revealed that establishing a common language about the software architecture is of crucial importance for successful architecture enforcement. Such a language is often project-specific, i.e., the software architects and developers define their own language with which they talk about the software architecture. Additionally, they use the language to describe crucial architecture rules associated with architectural design decisions so that they can be assessed subsequently by performing architecture conformance checking. In this chapter, existing conformance checking approaches are evaluated according to their suitability to formalize such project-specific languages. More precisely, the goal of this state-of-the-art analysis is defined as follows:

Goal: State-of-the-Art Analysis

To evaluate the ability of conformance checking approaches to define a project-specific language that can be used to 1) describe the software architecture and to 2) validate the architecture against its implementation.

In the following sections, the results of the state-of-the-art analysis are presented. First, the criteria based on the analysis goals are derived and explained in Section 4.1. Then, approaches are presented in separate sections and evaluated according to these criteria. Finally, the results are summarized in Section 4.8 and the need for a novel approach is motivated.

4.1. Criteria for Evaluating Related Work

Based on the analysis goal stated previously, the following criteria are defined according to which existing conformance checking approaches are evaluated:

Flexibility of the Modeling Language: The provided modeling language must be flexible in order to appropriately reflect the language used by the software architects and developers to describe and formalize the software architecture. Ideally, the approach provides mechanisms to extend the language with new elements so that concerns can be described and formalized that are not yet covered in the language.

This is needed due to the fact that existing approaches impose a language that is often not appropriate [WH05]. This means that languages provided by tools are typically restricted in terms of the architecture concepts and relations. The architect cannot extend the language by new concepts that are needed in the project as the tools are not flexible enough to represent the language with which software architects and developers talk about the architecture. They are forced to reformulate the project-specific architecture concepts using the available architecture concepts and relations of the specific tool, e.g. *module*, *component*, or *layer*, although such concepts might not be used in this project. Even if a tool's language might provide concepts with names that the architects need to describe the architecture, those concepts could have a different meaning than used in the project. Therefore, there is a high risk that the tool-specific language is not suitable capturing the intended architecture. In addition, even if the language could capture the architecture concepts and relations, it imposes an alternative vocabulary to the team that can lead to misunderstandings and ambiguities. Consequently, the original intentions of the architecture specifications and rules are prone to get lost. As a consequence, there is a risk that the language does not support to establish a common understanding of the software architecture (see activity "Achieving Mutual Understanding of the Architecture", Section 3.4.1). In order to be sufficiently flexible, the language should be customizable with user-defined architectural abstractions. This means that

- the vocabulary, i.e., names of architecture concepts and relations, of the language can be arbitrarily chosen and
- the semantics of concepts and relations can be defined unambiguously, so that it fits the purpose of the project. The semantics/meaning of architecture concepts describe which properties architecture concepts must have and how architecture concepts are (not) allowed to relate with each other.

Architecture Modeling Support: Architecture models are an important means for supporting architecture enforcement (see Chapter 3). Therefore, the approach should support the description of software architecture models, i.e., to express the application architecture based on the elements defined in the conceptual architecture.

Support for Architecture Documentation Integration: The intended architecture description should not be coupled to the tool infrastructure, so that the architecture description is accessible for anyone, especially for developers (see enforcement activity *Modeling Software Architecture For Developers*, Section 3.4.1). Ideally, the approach provides means to maintain the architecture description close to the source code, e.g., in the version control system. This also means that the architecture description can be documented with an arbitrary medium, such as Word documents or in a text file, as required by the project.

Understandability: Recent studies have revealed that approaches for (formal) architecture descriptions lack support for understandable architecture formalizations [ABO+17] [MLM+13]. As a result, architects and developers often neglect formal approaches. Some approaches provide support for informal natural language descriptions of the formalizations in order to clarify the intention of architecture rules to non-experts of the formalism. However, since the natural language is informal, it does not provide the unambiguity of the formalization. Consequently, the formalization and the informal description are prone to deviate from each other. An understandable formalization adds high value to architecture documentations. However, studies have shown that architecture documentations in practice are very long, complex, and not self-explanatory [OK13] [Ozk18b]. As a result, developers tend to avoid using the documentation as it does not provide the information developers need in order to perform their tasks [TBGH05]

[LSF03]. That is why, the approach should provide a modeling language that is usable and facilitates readable and understandable architecture descriptions, documentations, and formalizations to facilitate architecture enforcement.

The analyzed approaches are classified according to the categories *Reflexion-Model-based approaches*, *Rule-based Approaches*, *Logic-based Approaches*, *Query-based Approaches*, and *Embedded Specifications*. The categorization is based on a classification provided by [PTV+10]. The evaluation is complemented with an overview on related fields for software architecture descriptions, i.e., ADLs and UML. Figure 4.1 summarizes the results of the analysis. In the following sections, the detailed evaluation of the approaches is described.

4.2. Reflexion-Model-based Approaches

The reflexion model method [MNS95] specifies the intended architecture as a high-level model. This model contains architectural elements and dependencies that are expected between the elements. The model is used for comparison with the implementation model. For this, a dependency graph of the software system is extracted. In order to compare the high-level model and the implementation model, elements of the implementation model need to be mapped to elements of the high-level model. This mapping is created manually or with semi-automatic tool support [CKS05] [OEW18]. After performing the mapping, the differences between the models are calculated. The difference between the two is represented as the *reflexion model*. Edges in the reflexion model are classified according to the three categories:

Convergence: An element or relation that is allowed or was implemented as intended.

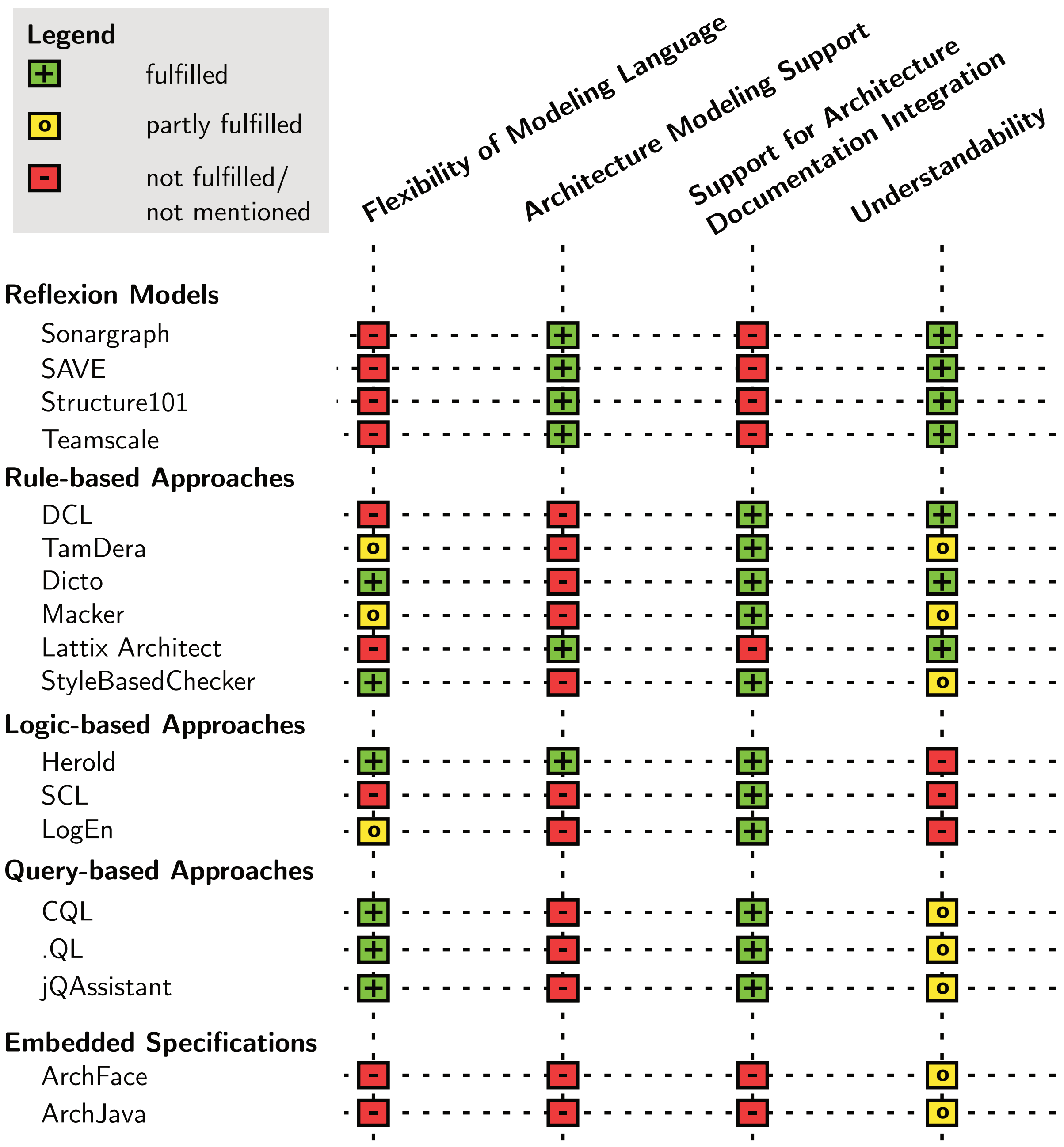
Divergence: An element or relation that is not allowed.

Absence: An element or relation that is missing in the implementation, i.e., that was intended but not realized.

The conformance checking tools *Sonargraph* [son17], *Software Architecture Visualization and Evaluation (SAVE)* tool [DKL09], *Structure101* [str18], and *Teamscale* [DHHJ10] implement the reflexion model method. For modeling the high-level model, they use a graphical modeling language that follows a well-defined meta model. The meta model defines the structuring elements the intended architecture consists of and the relations that connect the elements.

Figure 4.2 depicts the meta models of the modeling language of each tool. As can be seen, each modeling language provides different kinds of architectural elements the intended software architecture can be described with. For example, the tool Teamscale only provides one kind of element, i.e., component, whereas SAVE and Sonargraph each provide a more comprehensive meta model with different types of architectural elements that can be used to describe the intended architecture.

In Sonargraph, the software architect specifies the intended architecture in terms of *artifacts* that contain components. Components are top level programming elements, e.g., classes or interfaces. Artifacts define interfaces representing components other artifacts are allowed to use. Connectors are considered outgoing ports that are connected with interfaces of other artifacts. This means allowed and denied dependencies are specified by connecting the artifacts with each other or by prohibiting connections between artifacts, respectively.



Legend	
+	fulfilled
o	partly fulfilled
-	not fulfilled/ not mentioned

	Flexibility of Modeling Language	Architecture Modeling Support	Support for Architecture Documentation Integration	Understandability
Reflexion Models				
Sonargraph	-	+	-	+
SAVE	-	+	-	+
Structure101	-	+	-	+
Teamscale	-	+	-	+
Rule-based Approaches				
DCL	-	-	+	+
TamDera	o	-	+	o
Dicto	+	-	+	+
Macker	o	-	+	o
Lattix Architect	-	+	-	+
StyleBasedChecker	+	-	+	o
Logic-based Approaches				
Herold	+	+	+	-
SCL	-	-	+	-
LogEn	o	-	+	-
Query-based Approaches				
CQL	+	-	+	o
.QL	+	-	+	o
jQAssistant	+	-	+	o
Embedded Specifications				
ArchFace	-	-	-	o
ArchJava	-	-	-	o

Figure 4.1.: *Visualization of the results of the state-of-the-art analysis according to the defined evaluation criteria.*

With SAVE, the intended architecture is defined in terms of layers, subsystems, components, and clusters. It supports the programming languages Java, C++, and Delphi. A feature of SAVE is that conformance is checked continuously whenever changes are made to the source code. This ensures instant feedback about architecture violations in the source code.

Teamscale provides a architecture conformance checking functionality among other quality analysis services. The tool is implemented according to a pipe-and-filter architecture, so that new types of analyses, i.e., a filter, can be easily added for a new type of artifacts. This constitutes a crucial advantage compared to the other reflexion-model based approaches, since this adds flexibility to conformance checking. Other approaches do not support extension or it

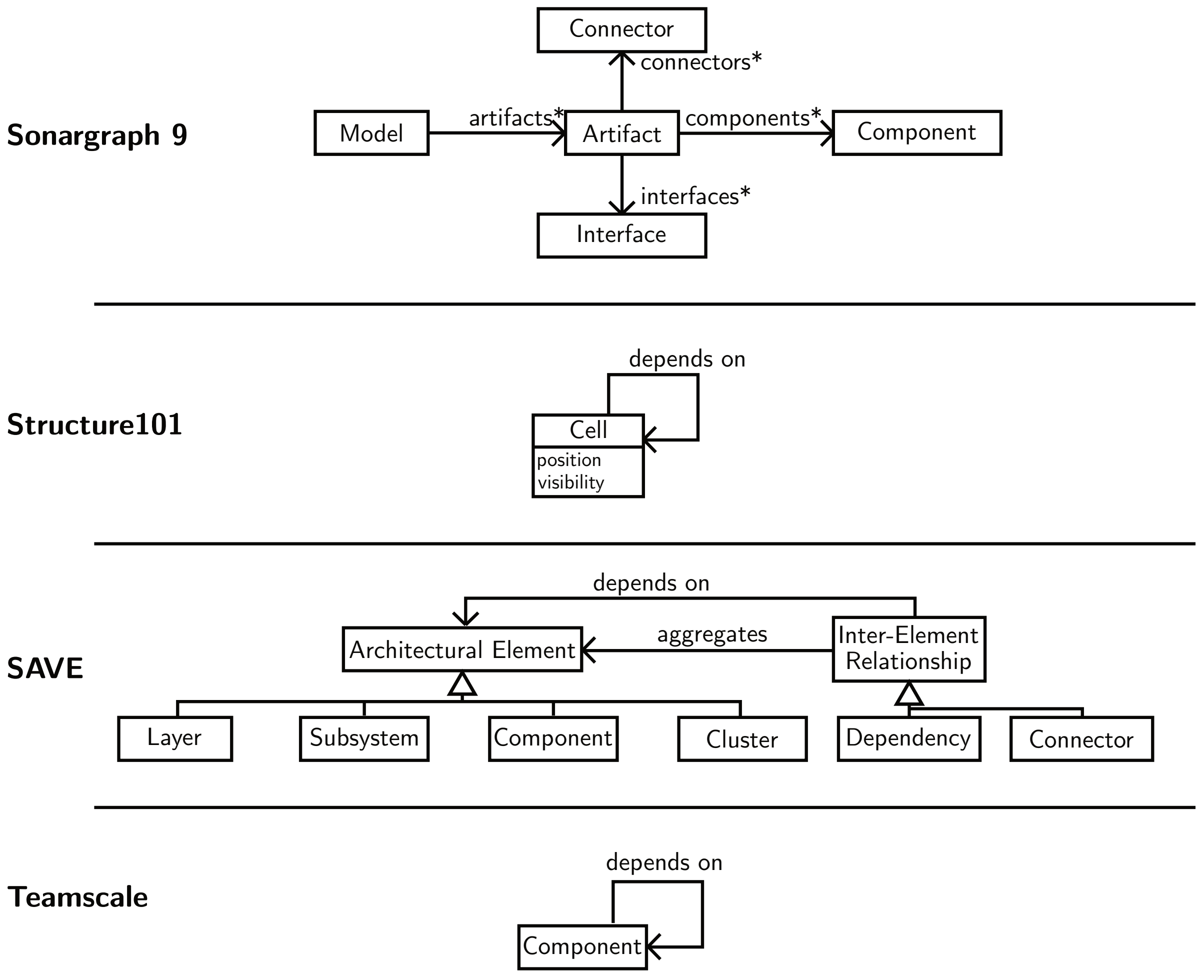


Figure 4.2.: *Meta models of conformance checking approach implementing reflexion modeling.*

is not clear how to extend the tool with analyses for other artifact types.

Evaluation

All presented reflexion-model-based approaches provide appropriate architecture modeling support. Additionally, by providing a graphical modeling notation, these tools support the understandability of architecture models and formalizations.

However, all approaches have in common that they are fixed in terms of their provided meta models defining the modeling language. This means that the modeling language cannot be easily extended with new language elements so that other aspects of the intended architecture can be described. All tools are specialized on formalizing and validating static dependencies rules. While those rule types are perfectly supported by the tools, formalizing more complex rules is either not supported or only hard to achieve as Prujit has shown in his experiment [PKB13]. Extending a tool with new language elements requires to change the language's tool infrastructure which is a tedious and high-effort process.

The intended architecture model is tightly integrated in the tools. It is not possible to integrate the description of the intended architecture into arbitrary types of architecture documentations. Although some tools like Sonargraph provide export functionalities, e.g., exporting XML-based representation of the architecture model, changes to the model still need to be applied in the tool and subsequently synchronized with the architecture documentation. This implies a high risk of the documentation and architecture model defined in the tool to deviate.

4.3. Rule-based Approaches

Rule-based approaches describe the intended architecture in terms of architecture rules. The approaches *DCL* [TV09], *Macker* [mac], *StyleBasedChecker* [Bec16], Dictō [CLN15], Lattix Architect [lat], and *HUSACCT* [PKvdWB14] provide a textual domain specific language (DSL) in order to define architecture rules in a separate document, e. g., in a text file. They allow software architects to specify constraints on static dependencies.

DCL is a declarative domain-specific language and allows for the specification of structural module dependency constraints. It provides *Module* as a language element which summarizes classes into a higher-level, architecture-abstraction. Architecture relations are defined on code level, e.g., *implements, accesses, throws* etc. DCL supports a number of rule types in order to express prohibitions (cannot), permissions (can), and obligations (must). Different types of dependencies (i.e., architecture relations) are provided by the language. Its syntax is very clear, however the expressiveness of the language is limited.

Macker uses an XML-based notation for defining architecture rules. XML-based descriptions can be enriched with a natural-language explanations in order to explain the actual intention of the rule, since the understandability potentially decreases with increasing complexity of the rule specification. Although the formalization can be enriched with natural language description, rule specification and its natural language representation tend to deviate from each other. As a result it could be possible that the natural language explanation does not represent the actual intention of the rule formalization.

The StyleBasedChecker [Bec16] supports the formalization of architectural styles. In this approach, an architectural style consists of typed architectural elements that are allowed to be related with each other or are not allowed to be related with each other. The architectural style is described in an XML-based language (similar to Macker).

Dictō aims for providing a unified specification interface for heterogeneous specification languages that are provided by tools, such as Moose [moo], PMD [pmd], or JMeter [jme]. Each rule triggers a specific validation tool that validates the specific rule. The advantage is that the software architect does not need to know about the specification language defined by the respective tool. He only needs to specify rules using the Dictō syntax.

Lattix Architect combines its graphical interface with a simple specification language for architecture rules (named *design rules* in Lattix). The dependencies between entities, e.g. classes, interfaces, package, of the system are visualized using a dependency structure matrix [SJSJ05]. In this approach, the software system is represented as a square matrix. The columns and rows of the matrix represent an element of the system, such as a subsystem, a module or a class. Each entry represents a dependency between two elements. The value 0 indicates that there does not exist any dependency between the elements. Any other numerical value denotes the strength of the dependency, i.e., how many dependencies exist between two elements. The specification language is similar to the one designed by DCL.

In their study, Pruijt et al. investigated existing conformance checking tools and validated their flexibility regarding rule formalization [PKB13]. They found that existing approaches failed in complex rule definitions. Based on their observation, they developed HUSACCT, a conformance checking tool that associates specific module types with rule types. For example, *layers* are specific module types that are associated with the *back call ban* rule type. This means that every instance of a layer is not allowed to call another layer above it.

Figure 4.3 shows exemplary formalizations using the languages DCL, Dictō, Husacct, and Macker. This example shows the formalization of a layer dependency rule. In this example, two layers *Model* and *View* are defined. *View* defines the top layer of the software architecture model, whereas the layer *Model* is below *View* in the layer hierarchy. This means that the layer *Module* is not allowed to depend on the layers above it including the layer *View*.

Evaluation

The rule-based approaches provide a clear syntax of their DSL resulting in understandable rule formalizations. However, most approaches are restricted in their meta model defining the language elements to express architecture rules. In order to customize the language, its tool infrastructure, e. g., the syntax and the corresponding parser, must be changed. Only Dictō provides a mechanism to extend the language with new rule types. Except HUSACCT and Lattix to some extent, no rule-based approach provides a means for explicit architecture modeling. The intended architecture is solely defined using architecture rules.

4.4. Logic-based Approaches

Architecture rules naturally map to predicates [Men00]. That is why, more sophisticated rule-based approaches make use of a logical formalism in order to formalize architecture rules. The approaches *LogEn* [MEM$^+$13], the *Structural Constraint Language* (SCL) [HH06], and the

Layer "Model" is not allowed to depend on the layer "View".

DCL

```
module Model: org.company.model.**
module View: org.company.view.**
Model cannot-depend View
```

Dictō

```
Model = Package with name:"org.company.model.*"
View = Package with name:"org.company.view.*"
Model cannot depend on View
```

Husacct

```
Model is not allowed to back call
```

Macker

```
<ruleset name="Model is not allowed to depend on View">
  <access-rule>
    <deny>
      <from class = "org.company.model.**">
      <to class = "org.company.view.**">
    </deny>
  </access-rule>
</ruleset>
```

Figure 4.3.: *Example formalizations using the rule-based approaches DCL, Dictō, Husacct, and Macker.*

approach suggested by [Her11] can be classified as logical-based specification approaches.

SCL defines first-order logic formulas that can be evaluated against the source code. It aims for capturing design intent as a constraint over object-oriented program structures for the programming languages Java and C++. As opposed to Herold's approach, predicates are defined on the source code structure and not on the architectural level. Predicates are predefined. SCL provides a declarative language similar to the Object Constraint Language (OCL) [WK03] that is used to define the rules.

LogEn allows to define so-called *ensemblies*. This allows source code elements to be combined to higher-level constructs. This provides a more coarse-grained view on the code. With the provided language, dependencies can be defined between ensemblies. The source code can then be checked for conformance according to this specification.

Herold presents an approach based on first-order logics [Her11]. He observes that current approaches lack flexibility in terms of the architectural aspects they are able to formalize and to validate. For example, most of the tools focus on the formalization and validation of structural dependency constraints or on a single architecture pattern, such as the Layered Pattern [BMR+96] (see Section 4.2). First-order logic provides an appropriate means to flexibly define architecture rules. As opposed to other approaches, the set of predicates – representing architectural abstractions – are not predefined and can be flexibly defined by the software architect or developer. This means that the architect is not restricted to a fixed meta model to describe the software architecture. Architecture rules are described as first-order formulas describing the properties a conforming system must have. Concrete software architecture models are described as facts based on the defined predicates.

Evaluation

Logic-based approaches have the advantage of being extremely extensible and expressive in terms of architecture description and formalization. As new concepts are needed for architecture descriptions, they can simply be added by defining new predicates. Due to this extensibility, it is possible to directly represent the vocabulary of the language used to describe the software architecture. However, those approaches are hard to use and to understand. Software architects and developers need to be experts in logical formalisms in order to define architecture rules or new language elements. Lozano et al. observe that developers need a more lightweight language that simultaneously allows for a flexible rule specification [LMK15]. The approach presented in this thesis can be classified as a logic-based approach with the advantage of being usable and understandable, since it provides a natural language interface for specifying architecture rules.

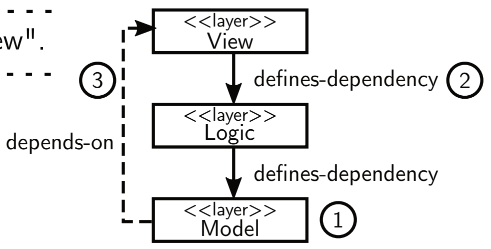
4.5. Query-based Approaches

Tools like the *Code Query Language* (CQLinq) [cql], .QL from Semmle [dMSV+08], and *jQassistant* [jqa] are query-based approaches. The main idea is to formulate queries based on the source code in order to retrieve single elements or more complex code structures. By doing this, several analyses on the source code can be performed, for example checking coding styles, finding bad smells, or even finding code structures that do not conform with architecture rules. Architecture rules are defined in terms of queries. An architecture rule is violated when a query result is not empty.

CQLinq is a code query language that has a similar syntax to the query language SQL. This means it uses `SELECT-FROM-WHERE` patterns to formulate queries. With this language, object-oriented source code can be queried. The language provides built-ins that are necessary to query object-oriented source code. Queries can be formulated in a very flexible way and the language is claimed to be easy to use. However, it is only possible to formulate queries on the code-level. Higher-level concepts on architecture level are not supported.

.QL from Semmle is similar to CQLinq, but allows for the definition of user-defined predicates that can be used in the queries. The syntax of the query language is close to Java, so that there is no need to learn a new language and no experts need to be involved in order to formulate queries.

jQassistant is based on the graph database Neo4J [neo]. This database provides a built-in query language named Cypher that is used to query the graph. When the source code is stored as a graph in the database, queries can be performed on it to find violations. A source code entity (e.g. class or method) is represented as a node in this graph. Dependencies between source code entities are named, directed edges between the nodes. Neo4J allows a user to enrich the nodes of the graph with additional labels to provide further conceptual information about the node. This can be used to add architectural information to a source code entity. For example, a node representing a Java class can have an additional label *Entity* that denotes the class representing the respective architecture concept. In order to add this information, Layer "Model" is not allowed to depend on the layer "View".



①

```
<concept id="Model">
   <cypher>
      MATCH
         model:Package
      WHERE
         model.name = "org.company.model"
      SET
         model:ModelLayer
   </cypher>
</concept>
```

Concept mapping for the layers View *and* Logic *is defined correspondingly with view.name = "org.company.view"*

②

```
<concept id="DefinedDependencyViewLogic">
   <cypher>
      MATCH
         view:Layer
      MATCH
         logic:Layer
      CREATE UNIQUE
         (view)-[:defines-dependency]->(logic)
   </cypher>
</concept>
```

Concept mapping for the dependency between Logic *and* Model *is defined correspondingly*

③

```
<constraint id="UndefinedDependency">
   <cypher>
      MATCH
         (layer1:Layer) -[:depends-on]->(layer2:Layer)
      WHERE NOT
         (layer1)-[defines-dependency]->(layer2) AND
         layer1.name <> layer2.name
      RETURN
         layer1.name, layer2.name
   </cypher>
</constraint>
```

Figure 4.4.: *Formalization of the rule* Layer "Module" is not allowed to depend on the layer "View" *using jQAssistant. The formalization shows the concept description of the Model layer and a definition of allowed dependencies between the two layers View and Logic. The last query shows the actual rule formalization.*

a corresponding query is used that states the conditions under which a class is labeled as a specific concept, e.g. using naming conventions. Figure 4.4 shows an exemplary query using the query language Cyper. It formalizes the layer dependency rule from Figure 4.3. First, the rule needs concept definitions that capture the layers *View*, *Logic*, and *Model*. Figure 4.4 illustrates how a (code-level) package with a specific name is mapped to the layer *Model*. Then, the allowed dependencies between layers are defined in another concept definition. Here, the allowed dependency between *View* and *Logic* is defined. The actual architecture rule is defined in the third query. In jQAssistant, this is called a *constraint*. This query returns all layers that have established a *depends-on* relationship (the formalization of this relation is not shown here), although no *defines-dependency* has been defined for those layers in a concept description, i.e., the *depends-on* relationship violates this layer dependency rule.

Evaluation

Query-based approaches provide a very powerful and flexible way to define and verify architecture rules. Due to their flexibility it is possible to cover a great variety of architectural aspects. However, query-based approaches are not capable of describing architecture models, i.e., the languages provided by the approaches are not suitable to express the building blocks of the software system. As can be seen in Figure 4.4, this often results in more verbose formalizations compared to rule-based approaches. The intention of the formalization is not directly clear. As a consequence, query-based formalizations tend to be less understandable.

4.6. Embedded Specifications

ArchFace [UNT10] and *ArchJava* [ACN02] embed architecture rule specifications directly in the source code. Architecture rules are specified as parts of code comments or annotations.

ArchJava provides additional programming language elements for Java that model architectural abstractions like components, ports, interfaces, and connections between the components. The approach aims for preserving the communication integrity [LVM95] between components. This means that components only communicate directly with other components as specified in the architecture. ArchJava validates the rules at build-time. ArchFace uses so-called *contracts* to specify architecture rules and validates them during runtime.

Both approaches aim for reducing the gap between software architecture and implementation by placing architecture specification to the code as close as possible. Architecture rules are therefore handled as a first-class entities during development activities allowing a shorter feedback loop. Developers are more aware when making architecturally-relevant changes that could affect crucial architecture rules.

Evaluation

Embedded specification approaches impose several drawbacks. First, extending the languages with new constructs is very cumbersome. For example, in the case of ArchJava, the syntax of the Java language needs to be changed in order to support additional architecture concepts and relations. This requires experts that are familiar with the Java syntax. Additionally, extending an existing programming language requires to extend the existing tool infrastructure, e.g., Integrated Development Environments (IDEs), in order to support new concepts. This again can only be conducted with great effort. All these approaches do not provide any mechanisms to appropriately extend the languages and the corresponding language infrastructure.

4.7. Architecture Description Languages

Formally describing software architecture is not only important for architecture conformance checking. The software architecture community recommends to use ADLs and UML for architecture descriptions and modeling. ADLs can even be used for tool-supported analyses, e.g., to verify software quality attributes based on architecture models. In the following, both modeling languages will be shortly presented and evaluated.

ADLs are not conformance checking approaches per se. However, they allow for a formal specification of software architecture which is a main prerequisite for conformance checking.

ADLs are formal, high-level languages that support a unambiguous description and analysis of software architecture. They describe software architectures in terms of components, their properties and connections among them. In their survey, Medvidovic et al. describe several ADLs for this purpose, as for example AADL [FGH06], xADL [DHT01], and Acme [GMW97] and classify and compare them.

Lago et al. define several features an ADL must provide based on a study they have conducted with practitioners [LMM+15]. Two important features mentioned are *extensibility and customization.* They state that practitioners need *"improved support for extending ADLs to better express domain- and project-specific concepts, for specifying constraints, and for enabling additional analysis capabilities"* [LMM+15]. However, ADLs still lack appropriate extension support. ADLs often impose a language that is not appropriate in the context of a project. As a consequence, architects need to restrict their architecture description to the concepts and relations provided by the ADL. This main challenge is addressed in the thesis by providing a formal means to flexibly define the language used to describe the software architecture and without being restricted to predefined, project-generic concepts.

As stated by [MLM+13] ADLs need to be simple and intuitive in order to communicate the architecture to the stakeholders that need information about the software architecture by simultaneously being appropriately formal in order to allow for various analysis tasks. However, a lot of ADLs often lack an appropriate level of understandability and usability and, consequently, often discourage architects and developers to use them. These are some of the reasons that ADLs, although being very powerful means to describe architectures, are still rarely used in practice as shown in empirical studies [MLM+13] [Ozk18b].

Architecture conformance checking is not supported by those languages. The main purpose of ADLs is to reason about the software architecture model that is completely disconnected from the implementation. However, some of them support architecture conformance by construction [PTV+10]. This means that the source code can be generated from the architecture model. For example, Darwin [MDEK95] supports the generation of skeleton code. Nevertheless, it is still necessary to validate the conformance when changes are made to the source code. Some ADLs can be enriched with constraints, but those constraints are only validated on the architecture model. Constraints are used in order to restrict the way how a component is allowed to be used and its semantics in terms of relationships and dependencies among internal elements of a component [MT00]. For example, the ADL Wright [All97] allows for the definition of architectural styles which may include constraints on the configuration of components.

4.7.1. Unified Modeling Language

UML [RJB04] [uml] is a standardized modeling language for software development. It is a graphical language providing a number of diagrams. Although it is not its primary purpose, it can also be used for software architecture modeling. For example, it provides component diagrams for showing the structural relationships between components of a system and deployment diagrams for modeling the mapping of source code artifacts to devices or software execution environments.

UML is still the most preferred language for modeling software architectures. As the study by [LMM$^+$15] has shown around 41% of developers in practice use UML exclusively, while only 12 percent of respondents use ADLs exclusively, around 35 percent mix an ADL and UML.

UML models can be enriched with constraints specified with the Object Constraint Language [ocl14]. Theoretically, UML and OCL together can be used for architecture conformance checking. However, UML also imposes – as ADLs – a restricted language with concepts and relations having a predefined meaning. The architecture needs to be adapted according to the constructs provided by UML. This is the same challenge that arises with ADLs.

UML provides so-called *profiles* that allows for extending the UML constructs. However, this implies that a corresponding tool needs to support new concepts and relations defined in this profile. A lot of effort must be invested in order to support new language constructs in existing tools [Vö10].

4.8. Conclusion

The state-of-the-art analysis has shown that existing approaches do not satisfy all the defined criteria.

Logic-based and query-based approaches perform best regarding expressiveness and flexibility, while the approaches of other categories are not sufficiently flexible and expressive to appropriately reflect the project-specific language. However, logic-based and query-based approaches often lack an understandable and usable modeling language.

That is why, a modeling language is needed that has the comparable high flexibility of logic-based approaches combined with a more usable and understandable architecture rule formalization that can be integrated into architecture documentations.

In the remainder of the thesis, an approach is proposed that applies *ontologies* as a flexible and formal means to capture and formalize the language the software architecture can be described with. In this thesis, this language is called *architecture concept language*. The approach additionally designs a CNL that is used as a frontend to define the architecture concept language and architecture rules. In this way, the need for flexible and understandable modeling languages is addressed.

5. Ontology-Based Architecture Enforcement

As motivated in the previous chapters, architecture enforcement is an important process in architecture-centric development. The results of the empirical study in Chapter 3 have emphasized that a shared understanding about the software architecture is a main prerequisite for successful architecture enforcement. For a shared understanding, a well established language about the software architecture is needed. Every team member must understand and use this language to achieve a consensus about relevant concepts in the project. In Chapter 4 it is revealed that state-of-the-art conformance checking approaches do not fully support architecture enforcement. Principally, this is due to the fact that existing conformance checking approaches impose a language that is often not appropriate [WH05]. This means that languages provided by tools are typically restricted in terms of architectural abstractions they provide. The architect cannot extend the language by new abstractions that are needed in the project. This means that languages provided by tools to describe the intended architecture are not flexible enough to represent the language with which software architects and developers talk about the architecture. In this thesis, this language is called *architecture concept language*. Consequently, the goal of this thesis is refined as follows:

Goal: Supporting Architecture Enforcement

To provide a means for architecture enforcement by supporting software architects and developers to flexibly define, share, improve, and enforce their architecture concept language throughout the development process.

The contributions of this chapter are the following: First, the term *architecture concept language* as it will be used and understood throughout the thesis will be defined more precisely (Section 5.1). Second, based on the findings of the study in Chapter 3, a refined definition of architecture enforcement is given (Section 5.2). In Section 5.3, an overview on an ontology-based approach for architecture enforcement is proposed. It is shown how the approach fulfills the goals and criteria defined in Chapter 4 and the main parts of the approach are presented. Finally, an overview on related work of ontology-based approaches in software engineering is presented in Section 5.4.

5.1. Architecture Concept Language

As described in Section 2.1, software architecture also serves as a language that is established for communication between different stakeholders. During the architectural synthesis phase, the software architect creates an architecture solution that comprises the *conceptual architecture* and the *application architecture* (Section 2.1). In this section, it is explained how the conceptual architecture can be understood as a language used by software architects and developers to talk about the software system.

In [Vö10], Völter emphasizes that it is crucial to have a consistent language to talk about the software system. This language is manifested in the conceptual architecture that is created during the architectural synthesis phase. The conceptual architecture contains a platform-independent terminology with which the software architecture is described. Software architects and developers use terms that describe core architectural abstractions and their related responsibilities and properties. Terms implicitly define semantics of these abstractions. The process of defining a conceptual architecture helps the development team to better define, understand, and clarify architectural abstractions that are necessary to define the software architecture. By using terms for architectural abstractions of the conceptual architecture, software architects and developers implicitly decide for a language [Vö10] to talk about the software architecture. This means, they use terms denoting *architecture concepts* and *architecture relations* with which they describe the software architecture. An architecture concept is defined as follows:

Definition 5.1.1: Architecture Concept

An architecture concept represents a specific type of a core architectural abstraction with well-defined properties. It has a well-defined semantics and is described by a term.

Architecture concepts are related with each other by *architecture relations*:

Definition 5.1.2: Architecture Relation

An architecture relation is a named relationship that connects two architecture concepts.

Examples of architecture concepts are *component*, *interface*, *pipe*, *filter*, or *layer*. The relation *provide* connects the concepts *component* and *interface*: a component provides an interface. Architecture concepts have properties that are defined by *architecture rules*. Every instance that is classified as being a member of a specific concept must satisfy its corresponding properties, i.e., must conform with the architecture rules. More precisely, an architecture rule is defined as:

Definition 5.1.3: Architecture Rule

An architecture rule prescribes which architecture relation an architecture concept is allowed to have (permission), must have (obligation), or is not allowed to have (prohibition) with another architecture concept.

The language that software architects and developers use to describe the architecture is called *architecture concept language*. An architecture concept language has two important characteristics: First, it has limited generality. It defines only the concepts and relations that are needed to describe the software architecture and that are specific to it. That is why, it is considered as a DSL. Second, it is a formal language allowing the meaning of architecture concepts to be defined unambiguously. Unambiguity is important so that every stakeholder in the team understands the meaning of the respective architecture concept. Additionally, a formal definition of abstractions allows for sophisticated, tool-supported analyses, such as validating the conformance of the source code to the rules defined by the architecture concept language.

In this thesis, it is defined as follows:

Definition 5.1.4: Architecture Concept Language

An architecture concept language is a formal, domain-specific language defining the architecture concepts and architecture relations software architects and developers need to describe the software architecture of a software system. More precisely, the architecture concept language is a tuple $ACL = (AC, AR, R)$ where

- AC is the set of architecture concept names,
- AR is the set of architecture relation names, and
- R is the set of architecture rules.

5.2. Architecture Enforcement Revisited

In Chapter 2, a general view on architecture enforcement is given. Figure 2.1 illustrates how architecture enforcement integrates in the phases of the architecture-centric software development process. It shows that the goal of architecture enforcement is to ensure that the implementation follow the conceptual architecture. Additionally, the results of the study in Chapter 3 revealed that a shared understanding of software architecture is one of the main prerequisites for successful architecture enforcement.

Consequently, the corresponding understanding of architecture enforcement as used throughout this thesis is as follows:

Definition 5.2.1: Architecture Enforcement

Architecture enforcement is the process of establishing and explicitly capturing a common architecture concept language that is used by all members of the software development team on a daily basis and that is visible in the source code as well as the documentation. The architecture concept language provides an unambiguous way to describe architecture concepts and relations, as well as the rules that apply to them. Additionally, architecture enforcement encompasses the verification whether the implementation conforms with the architecture concept language.

Defining a common architecture concept language to capture the important concepts and their properties should support architects and developers to better understand and clarify their software architecture. The process of defining the language actually helps the software architects and developers to better understand, clarify, and refine architectural abstractions, i. e., architecture concepts and relations. Having a consistent language about the software architecture is an important prerequisite for successful architecture enforcement (see Chapter 3). For this, methods are needed to precisely capture and share the language during development and maintenance. Additionally, the language should be used for validating rules defined by the language against the implementation to ensure architecture conformance.

As elaborated further in Section 5.3.1, ontologies and description logics serve as a natural foundation for formalizing architecture concept languages and provide the required formal

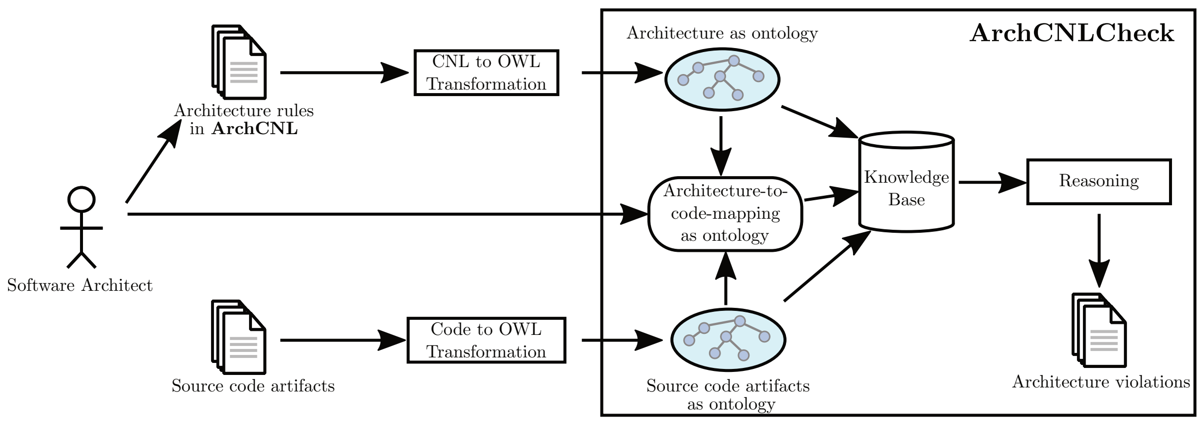

ArchCNLCheck
Architecture rules in ArchCNL
CNL to OWL Transformation
Architecture as ontology
Software Architect
Architecture-to-code-mapping as ontology
Knowledge Base
Reasoning
Source code artifacts
Code to OWL Transformation
Source code artifacts as ontology
Architecture violations

Figure 5.1.: *Overview of the approach.*

framework and tool-support to verify the language against the implementation. That is why, this thesis proposes *ontology-based architecture enforcement*, i.e., an approach that leverages ontologies and description logics to support architecture enforcement (Definition 5.2.1). An overview of the approach is described in the next section.

5.3. Overview of the Approach

The proposed approach and its constituent parts are depicted in Figure 5.1. The approach uses ontologies to formally define the architecture concept language, to represent source code, and to define the architecture-to-code mapping in a unified way. This means that each required artifact (software architecture, source code, mapping) is described and stored using the same underlying representation, namely ontologies.

The approach provides two components: First, it provides a specification interface that is used by the software architect for defining the architecture concept language. The specification interface is implemented by a CNL that is called the *Architecture Controlled Natural Language (ArchCNL)*. Second, it provides *ArchCNLCheck* which is responsible for verifying whether the architecture rules defined in *ArchCNL* are respected in the source code. *ArchCNLCheck* implements dedicated scanners that allow for a tool-supported transformation. It makes use of reasoners (see Section 2.4.4) to reveal violations against the architecture rules defined by the architecture concept language.

Figure 5.2 visualizes the detailed steps of the approach. In the following, these concrete steps of the approach are described.

Basically, the approach comprises two phases, whereas each phase is further divided into several steps. The first phase of the approach concerns the *definition of the architecture concept language*. In the second phase, the actual *architecture conformance checking* is performed. More specifically, the following steps are performed:

Step 1: Specify architecture rules as *ArchCNL* sentences In this step, the software architect formally defines the architecture concept language. For this, the software architect makes use of *ArchCNL* that acts as a natural language interface built upon the description logics formalism. *ArchCNL* provides a more usable and understandable way to define the language and architecture rules. The architecture concept language is defined by

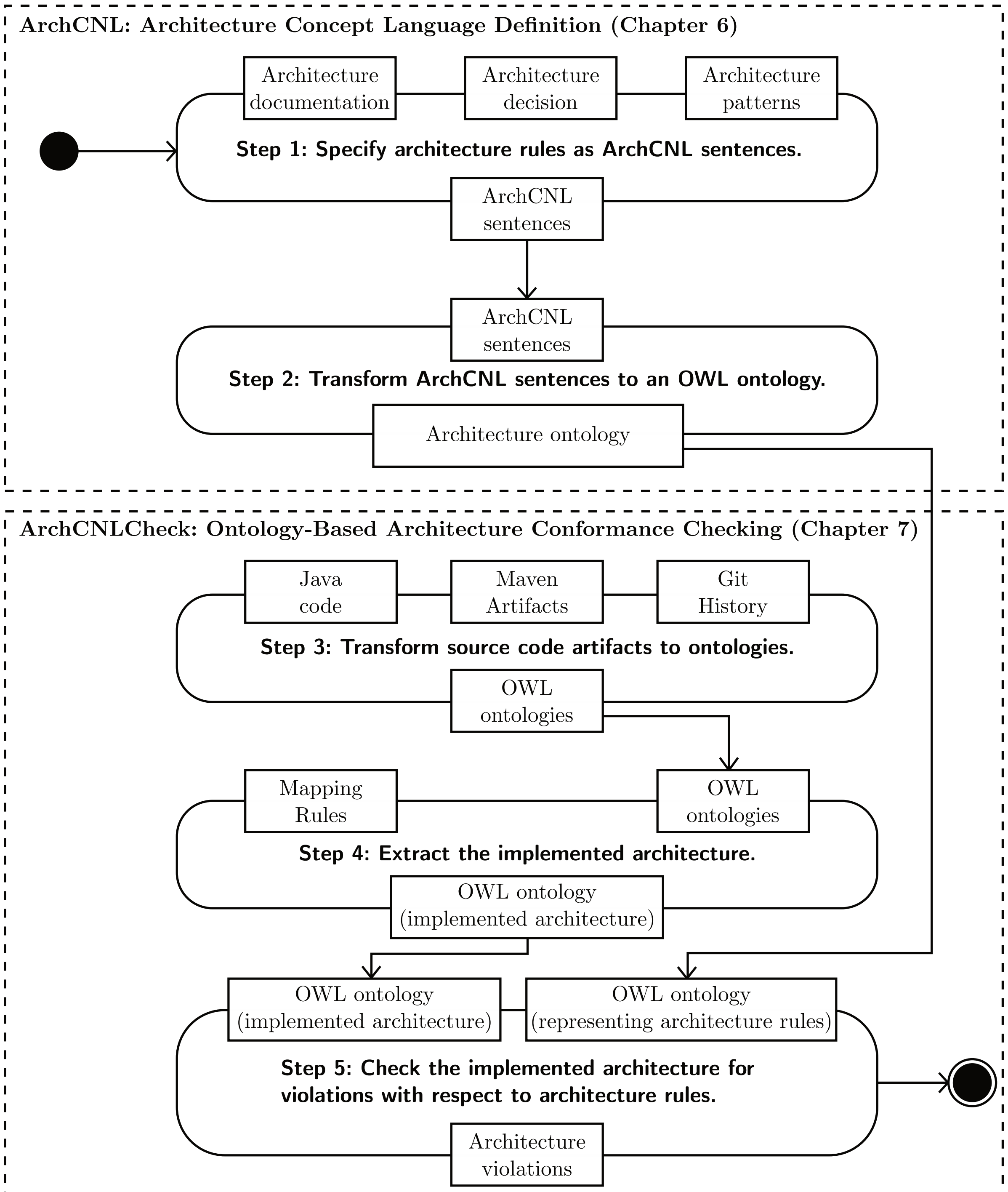

ArchCNL: Architecture Concept Language Definition (Chapter 6)
Architecture documentation
Architecture decision
Architecture patterns
Step 1: Specify architecture rules as ArchCNL sentences.
ArchCNL sentences
ArchCNL sentences
Step 2: Transform ArchCNL sentences to an OWL ontology.
Architecture ontology
ArchCNLCheck: Ontology-Based Architecture Conformance Checking (Chapter 7)
Java code
Maven Artifacts
Git History
Step 3: Transform source code artifacts to ontologies.
OWL ontologies
Mapping Rules
OWL ontologies
Step 4: Extract the implemented architecture.
OWL ontology (implemented architecture)
OWL ontology (implemented architecture)
OWL ontology (representing architecture rules)
Step 5: Check the implemented architecture for violations with respect to architecture rules.
Architecture violations

Figure 5.2.: *Overview of the steps that need to be performed, the required input of each step and the output created by each step.*

specifying architecture rules as sentences written in *ArchCNL*. Sentences represent architecture rules and implicitly define architecture concepts and relations. Terms and the meaning of architecture concepts and relations can be driven by terms used in architecture documentation, architecture decisions, chosen patterns, styles, or reference architectures etc. This step is performed by the responsible software architects or developers. The syntax of *ArchCNL* is described in Chapter 6.

Step 2: Transform *ArchCNL* sentences to an OWL ontology In a next step, the approach automatically transforms the sentences written in *ArchCNL* to an OWL ontology that comprises concepts, relations, and class axioms. This means that the architecture concept language defined by the architect is represented as an ontology called the *architecture ontology.* This process is performed automatically. The corresponding algorithm extracts the concepts and relations used in the *ArchCNL* sentences and converts them to concepts and relations of the description logics formalism. *ArchCNL* sentences are transformed to class axioms.

Step 3: Transform source code artifacts to ontologies The class axioms are applied on the implementation in a later step in order to verify whether the implementation adheres to the architecture concept language. That is why, it is necessary to transform the implementation to an ontology-based representation on which the class axioms can be applied.

The implementation normally consists of several, heterogeneous types of source code artifacts. The gap between different types of source code artifacts is bridged by uniformly representing them as ontologies. The ontologies model the most important concepts and relations of each artifact type. For example, the approach provides ontologies to represent Java source code, Maven artifacts, and history data from Git [git] repositories. Dedicated parsers automatically transform the artifacts to individuals of the respective concepts and relations. Note that any artifact can be modeled in this way. That is why, the approach provides a flexible way to check arbitrary types of artifacts for conformance. The facts are then imported to a knowledge base on which the next steps of the process are performed, e.g., consistency reasoning. In Section 7.2 the ontology design for source code artifacts is elaborated in more detail.

Step 4: Extract the implemented architecture The software architect defines the architecture-to-code-mapping by *mapping rules* that describe how the implemented architecture should be extracted from the source code artifacts. These mapping rules provide a recipe for extracting architecture concepts and relations from the code and the ontology-based representations of the source code artifacts from the previous step.

The mapping rules could be based on naming conventions, inheritance relations, or direct specifications of concrete code elements. The approach is open for various ways to specify the mapping. In any case, a mapping has to exist in some way, since it is one of the major goals of architecture enforcement to make the architecture visible in the code (see Chapter 3).

The mapping rules are applied on the ontology-based representation of source code artifacts using reasoning (see Section 2.4.4). As a result, the ontology-based representation of source code is enriched with architectural information, i.e., instances of code concepts are assigned to architecture concepts and code relations are assigned to architecture relations of the architecture ontology.

The implemented architecture is again represented as an ontology. This ontology combines code-level as well as architecture-level concepts and relations. In Chapter 7 it is explained how mapping rules are formalized as SWRL rules.

Step 5: Check the implemented architecture for violations with respect to architecture rules The results from step 4 (implemented architecture) and step 2 (architecture rules) serve as input for this final step. Class axioms that have been transformed previously from the sentences written in *ArchCNL* can be applied on the implemented architecture. For this, reasoning (see Section 2.4.4) is applied that checks whether the individuals representing the implemented architecture are consistent with the class axioms of the architecture ontology, i.e., with the architecture rules. Inconsistencies represent architecture violations which in turn are the final result of the whole conformance checking process.

5.3.1. Recapitulation of Criteria

In the following, it is discussed how the approach fulfills the criteria described in Chapter 4.

Flexibility of the Modeling Language: Ontologies and description logics serve as a natural foundation for conveying architecture concepts and relations. At the same time ontologies and description logics provide a rigorous semantic framework for reasoning about the consistency of the language and for verifying whether the language is consistently used in the implementation. Generally, ontologies are used to capture the most important concepts of a domain and to establish a common understanding about these concepts among stakeholders. Ontologies are therefore a natural fit for representing and formalizing an architecture concept language. Since ontologies are not restricted to specific concepts and relations, the architect can define architecture concepts and relations as they are needed for the project. By providing flexibility, the language does not impose a priori restrictions on the architecture concepts and relations that can be considered for describing the software architecture and corresponding rules. The software architect can define arbitrary architecture concepts as needed without being restricted by existing concepts and the meaning of architecture concepts can be defined as needed. In this way, the architecture concept language can be easily extended with new concepts and relations if necessary, existing concepts and relations can be removed, or existing concepts and relations can be refined. Therefore, the software architect does not need to adapt his mental model to the actual language imposed by a modeling language. Consequently, this reduces the semantic gap between his mental model and the conceptual architecture. Since ontologies provide the required expressiveness and flexibility for defining an architecture concept language, this criterion is fulfilled.

Architecture Modeling Support: As described in Section 2.4, description logics structure a knowledge base into the TBox and the ABox. The TBox corresponds to the conceptual architecture containing the architecture concepts and relations the application architecture can be described with. Correspondingly, the ABox defines the instances of these concepts and relations referring to the application architecture. In this way, description logics and ontologies allow architects and developers to express the conceptual and application architecture in the same formalism. That is why, the proposed approach provides architecture modeling support which is an important means to support architecture enforcement.

Support for Architecture Documentation Integration: *ArchCNL* is a textual DSL. This has the advantage that descriptions written in *ArchCNL* can be easily embedded into architecture documentations. In this way, architecture models can be enriched with text-based descriptions. Furthermore, informal, text-based architecture documentation templates can be enhanced with formal and therefore verifiable sentences written with *ArchCNL*. In this approach, a tool chain is developed that allows for integrating *ArchCNL*-based descriptions into asciidoc-based documentation templates such as for arc42 [arca] and ADRs templates [adr].

Understandability: For OWL, a lot of notations have been proposed that aim to be understandable and usable for ontology authors and users. For example, the OWL Manchester syntax is a user-friendly syntax to describe OWL ontologies [man12]. Other approaches aim for describing ontologies in natural language sentences using CNLs that have been proven to be understandable and usable in the context of ontology authoring [Kuh13] and that can be quite straightforwardly mapped to description logics and OWL [DCFKK09]. *ArchCNL* is based on this type of CNLs. Therefore, the approach potentially offers an understandable means for architecture rule formalization.

Besides those criteria, using ontologies for architecture enforcement implies additional advantages. It should be emphasized that description logics and ontologies are defined by a formal syntax and semantics. That is why, the meaning of concepts and relations of the architecture concept language can be defined unambiguously. Additionally, it comes with expressive reasoning power that is decidable. Mature and efficient reasoning services can be exploited in order to check the architecture concept language for inconsistencies and to automatically validate whether architecture rules are violated.

Furthermore, since OWL represents a uniform data format that does not presume a specific domain, any domain can be described. This means, any type of data can be described such as the architecture model, design models, and also source code artifacts while using the same underlying representation. These heterogeneous types of artifact can be stored uniformly into one knowledge base.

A lot of tools for ontology authoring and manipulation have been developed, such as Protégé [Mus15] [pro]. They provide editors for creating and changing ontologies, visualizing them, and to reason on them. When ontologies change, the tools do not need to be changed in order to visualize ontologies, since those tools always work on the same underlying representation. These tools can also be used to visualize and manipulate the architecture concept language.

Description logics are the formal basis for ontology languages and for Semantic Web [sem] technologies (see Section 2.4). For the Semantic Web, a lot of mature standards such as OWL or the SWRL have been developed. That is why, rich tool support is available to implement the approach. OWL constitutes a standardized and interoperable format to store ontologies. It provides a common notation that is not predisposed to any particular language (such as ADLs or UML). This facilitates the sharing and reusing of the architecture concept language. Moreover, ontologies can be defined in modules. Each module represents an architecture concept language. Modules can be imported into existing ontologies and therefore be reused and extended. Moreover, ontology modules can be combined. A software system can then be verified to which extent it conforms to specific language modules. For example, an architecture concept language for Service-Oriented Architecture (SOA) [Erl05] could be derived from an architecture concept language that models component-based architectures: A software system that implements a SOA must not only satisfy the rules formulated in the SOA language, but also those defined in the architecture concept language defining component-based architectures. The SOA language inherits all the architecture concepts and associated rules from the architecture concept language containing concepts and relations of component-based architecture and adds more rules that need to be satisfied by the respective software system. In this way, it can be validated to which extent a software system satisfies the constraints of each architecture concept language. Reusing architecture concept languages aims to increase the efficiency of defining new languages. A repository of existing architecture concept languages can be created from which a software architect can choose the most appropriate ones.

As can be seen, ontologies provide a promising means to flexibly formalize and unambiguously define architecture concept languages and to automatically verify whether the language is consistently used throughout the project.

5.4. Related Work on Ontology-Based Approaches in Software Engineering

Ontologies and description logics are widely adopted in software engineering. However, there is no approach applying these formalisms to flexibly define architecture concept languages and to use this language to verify the conformance between the intended architecture and the implementation in order to support architecture enforcement. In the following, related work on ontology-based approaches will be presented.

Ontologies have attracted attention in a lot of applications of software engineering. For example, the Semantic Web is part of the research agenda of the Knowledge-based Software Engineering community, e.g., in the Software Engineering and Knowledge Engineering [sek]. Ontology-based approaches exploit ontologies for nearly all crucial activities of software development, e.g., requirements engineering, architecture documentation, pattern formalization, consistency checking, and model-driven engineering. Happel et al. provide a comprehensive overview on ontology-based approaches in software engineering [HS06].

Capturing architecture knowledge in form of ontologies has been explored by Kruchten and Farenhorst et al. In [Kru04a] Kruchten suggests an ontology for architectural design decisions. In [FdB06] and [DBFL$^+$07] Farenhorst et al. define a model of architecture knowledge that can be mapped to ontologies.

The approach proposed by Graaf et al. is closely related with the approach presented in this thesis. Graaf et al. propose ontologies for describing and preserving architecture knowledge. As opposed to file-based approaches, an ontology-based documentation approach allows for effectively locating and retrieving architecture knowledge as the authors have shown in [dGTLvV12]. Additionally, having an ontology-based representation of architecture knowledge allows for sophisticated analyses in order to reveal implicit relationships between knowledge elements in architecture documentation. Graaf et al. propose a lightweight ontology containing concepts and relations representative for architecture knowledge. Architecture knowledge is stored in a semantic wiki. The wiki provides Semantic Web technologies in order to store and retrieve the required information [dGTLK17] about the respective software architecture. However, the approach is not designed to verify the documentation against the design and the implementation and is solely restricted to documenting software architecture. In contrast, the approach proposed in this thesis supports both documentation and architecture conformance checking.

Ontologies and description logics have also been applied for consistency checking in UML diagrams. The advantage of applying description logic on UML is that the formalism adds formal semantics to UML models and therefore complements the UML specification with verifiable consistency rules that are not enforced by the UML specification itself. For example, the approach presented by [VDSMSJ03] applies description logics reasoners to reveal inconsistencies in UML diagrams. For this, UML diagrams are transformed to individuals based on predefined transformation rules. These rules define how elements of UML diagrams have to be transformed to corresponding facts of the description logics knowledge base. As mentioned by the authors, the approach is able to detect inconsistencies on different levels, i.e., at the model level, the instance level, and between the model and the instance level.

In [SB05], the authors describe an approach to verify architectural features in component diagrams by applying description logics. They implement a tool that provides different types of consistency checks that verify whether components and interfaces are consistently defined. For example, it is verified whether classifiers, e.g. classes, belonging to different components are only indirectly associated through the required and provided interfaces of the respective components.

Ontologies and description logics are used to formalize architecture patterns, such as presented in [PGH09] and [CSW18]. The approaches apply the formalisms to develop an ontological approach for architectural style definition and style combination. Using ontologies, they formalize main concepts and relations of architectural styles and apply reasoners to verify whether style definitions are consistent. Having such a unified, ontological representation of architectural styles allows for combining different styles with each other to define new kind of architectural styles. However, as opposed to the approach presented in this thesis, the approaches do not verify whether an implementation conforms with the ontology-based pattern formalization, since consistency checking is only applied on an architectural level. Nevertheless, verifying the conformance between architecture and code is a crucial activity in the context of architecture enforcement (see Chapter 3).

Ontologies are also adopted in the context of model-driven engineering and DSLs. For example, Staab et al. combine model-driven engineering with ontology technologies [SWGP10]. They define an ontology-based meta model as a core means to exploit ontology technologies for software modeling. The combination of model-driven engineering and ontologies is implemented in the TwoUse framework developed by Parreiras [Par11].

Walter et al. propose an approach to describe DSLs with ontologies [WSPS09]. They develop the framework *OntoDSL* that exploit the formal semantics of OWL together with reasoning services. Applying ontologies and description logics in DSL development allows for a precise and formal definition of domain concepts and their semantics in an extensible way. By applying reasoners, the correctness or the satisfiability of defined concepts of a DSL can be verified.

Other approaches use ontologies to describe different types of source code artifacts, such as source code, repository data, or bug tracking data, in a unified way. SEON is a family of software evolution ontologies [WGH$^+$12] structured as a pyramid of ontologies. The goal of the approach is to *"provide a taxonomy to share software evolution data of various abstraction levels across the boundaries of different tools and organizations"* [WGH$^+$12]. The ontologies are described with OWL. The ontologies describe main concepts and relations prevalent in software evolution, such as stakeholders, their activities, and artifacts that are created during evolution and how they are related with each other. Having formalized the software evolution domain as ontology allows evolution data to be used for querying, reasoning, and exchanging. Relationships between different types of artifacts can be explicitly described and queried. The pyramid contains ontologies for describing source code, data from version control systems, bad smells, or data from bug tracking.

EvoOnt [TKB10] is a similar, more lightweight approach. It contains three ontologies used to describe object oriented systems, bugtracking meta data, and repository data from version control systems. Ideas from [WGH$^+$12] and [TKB10] are adopted in this thesis to represent source code artifacts as ontologies for the ontology-based conformance checking process.

5.5. Conclusion

In this chapter, an ontology-based approach for supporting architecture enforcement has been proposed. It consists of two components, namely *ArchCNL* which is a CNL used to define the architecture concept language of a project and *ArchCNLCheck* that which uses the specified architecture concept language to verify the intended architecture against the implementation. The term *architecture concept language* has been defined and the understanding of architecture enforcement as it will be used in this thesis has been refined based on the result of the empirical study in Chapter 3 and of the state-of-the-art analysis in Chapter 4. Additionally, the criteria defined in Chapter 4 have been recapitulated and discussed with respect to the proposed approach. The discussion demonstrates that the approach could provide a promising way to support the architecture enforcement process. In the following chapters, the approach will be described in more detail. In Chapter 6, the syntax and semantics of *ArchCNL* to specify the architecture concept language is presented. Subsequently, *ArchCNLCheck* is presented in Chapter 7.

Related work on ontology-based software engineering has been presented and compared to the proposed approach. It has been shown that there is no approach that allows for defining and validating the architecture concept language against the implementation. Nevertheless, since all the presented approaches are based on the same formalisms, namely description logics and ontologies, the proposed approach could be easily integrated with them. Existing approaches could be combined in a comprehensive methodology and advantages and benefits of these formalisms, as elaborated in this chapter, can be efficiently exploited in the software engineering process.

6. Ontology-Based Definition of Architecture Concept Languages

As described in the previous chapter, having a consistently defined language about the software architecture is a crucial prerequisite for successful architecture enforcement. That is why, an ontology-based approach is proposed that formally captures this language by means of ontologies and description logic. In this chapter, it is described how architecture rules can be used to define such a language and how these rules are formalized with class axioms (phase 1 of the approach, see Figure 5.2). The approach is supported by the *Architecture Controlled Natural Language (ArchCNL)*, a CNL that provides a more intuitive way to interact with the formalism. By writing natural language sentences, the software architects and developers define the architecture concept language. These sentences can be automatically transformed into an OWL ontology. In a next step, these sentences are used for tool-supported analyses in a next step (phase 2 of the approach, see Figure 5.2), namely architecture conformance checking (Chapter 7).

6.1. ArchCNL: A Controlled Natural Language to Define Architecture Concept Languages

In Chapter 5, the need for a so-called architecture concept language was motivated. Such a language defines architecture concepts used to describe the software system on an architectural level as well as the architecture relations between these concepts. One major aspect of the approach is to use ontologies and description logics in order to capture and formalize architecture concept languages. This provides the following benefits:

1. The ontology provides an accurate documentation of the architecture concepts and relations that can be shared among the team members.
2. The meaning of concepts is defined unambiguously.
3. Architecture rules of architecture concepts are defined formally. This facilitates tool-supported analyses, such as evaluation of the consistency of the language itself and validation of the consistency of the implemented architecture with respect to the defined rules (i.e., architecture conformance checking, see Chapter 7).

Basically, architecture concepts are mapped to concepts in description logics and architecture relations are mapped to roles in description logics. Consequently, architecture rules can be formalized as class axioms. The description logics constructors – e.g. existential, universal, or cardinality – then provide the necessary means to express rule semantics. The rule semantics determine whether the architecture rule describes a permission, prohibition, or an obligation.

As described in Section 2.3.2, architecture conformance checking approaches aim for detecting divergences and absences. That is why, the approach must support the corresponding *architectural rule types* in order to detect these types of violations. The approach defines such rule types based on the semantics given by the constructors of the description logics formalism. In Section 6.3, the corresponding type of violation (absence or divergence, see Section 4.2) that can be detected with the respective rule type is defined. These rule types can be commonly found in architecture documentations as observed by [CLN14].
Unfortunately, the description logics formalism requires experts to formulate architecture rules as class axioms. Few software architects or developers are familiar with this formalism. That is why, formulating architecture rules as class axioms can be quite cumbersome and unusual. In order to support a more understandable and usable specification language, *ArchCNL* is proposed. It is a CNL that acts as a virtually natural language interface of description logics. *ArchCNL* provides a more intuitive way to describe rules compared to the syntax of description logics. In the following, the syntax and semantics of the language is described. The language design is driven by the fact that specific architectural rule types can be classified based on the semantics of the constructors. Based on these rule types, the grammar of *ArchCNL* is defined.

6.2. Design Decisions and Syntax of the Controlled Natural Language

In this section, *ArchCNL* is presented that hides the formalism behind a more understandable natural language interface used to specify architecture rules for architecture concepts and relations. The main idea of *ArchCNL* is to describe architecture rules for architecture concepts as natural language sentences. By writing natural language sentences, the software architect defines the architecture concept language.

First, general design decisions of *ArchCNL* are given. After that, the grammar of *ArchCNL* is presented. This includes the general structure of architecture rules written in *ArchCNL* and its vocabulary.

6.2.1. General Design Decisions for the CNL

The fundamental decision needed for *ArchCNL* is on which natural language it should be based on. English is used as a base language, since English is a common language spoken in software engineering and is therefore widely understood by software engineers. This enables *ArchCNL* to be applicable by a wide range of users in the software engineering community.

Design decisions from existing CNLs are reused that act as a natural language layer for ontology languages, such as OWL and description logics [DCFKK09, Sch02]. They aim to facilitate the ontology authoring process by hiding the complexity of the formalism used for modeling ontologies. Several controlled natural languages for ontology authoring exist, such as Attempto Controlled English [DCFKK09] or RabbitOWL [SKC$^+$08]. As architecture rules are formalized with description logics in this approach, some design decisions of those CNLs can be applied to *ArchCNL* as well.

However, CNLs for ontology authoring provide a more comprehensive grammar than needed in the context of architecture rule formalization. *ArchCNL* is restricted to support the architectural rule types presented later (Section 6.3). This helps to establish a bijective mapping between the rule types and their corresponding representation in description logics. Thus, for each rule type a specific sentence structure is specified.

It is important to take the trade-off between the naturalness and the closeness to the underlying formalism of *ArchCNL* into consideration. A key decision of *ArchCNL* is to make it as natural as possible, so that architecture rule formalizations are readable and understandable. This means that *ArchCNL* should avoid to use keywords that are close to the underlying formalism and that are consequently not known by nonspecialists. *ArchCNL* should enable a software architect to focus on architecture rule formalization without being confused by concepts specific to the underlying formalism. Thus, appropriate expressions need to be found that maintain the naturalness of the sentences. At the same time it is necessary to avoid that the sentences written with this language become open to interpretations.

6.2.2. General Sentence Structure

The grammar of *ArchCNL* is depicted in Grammar 6.1. A specification of an architecture concept language is a collection of so-called *ArchCNL* sentences. Such a specification contains at least one sentence, i.e., an empty specification is not allowed. An *ArchCNL* sentence is defined as follows:

Definition 6.2.1: ArchCNL Sentence

An *ArchCNL* sentence is a sentence that follows Grammar 6.1. An *ArchCNL* sentence represents an architecture rule.

Basically, an *ArchCNL* sentence follows the pattern:

subject concept expression + modifier + object concept expression + .

The subject concept expression is a noun phrase and refers to the architecture concept for which a rule is defined. The subject is always introduced by a keyword, as for example *Every*, *No*, or *If*. The modifier defines the rule semantics. Currently, the keywords *can*, *must*, *only* are supported. The object concept expression is a verb phrase. It is realized by a verb that represents the architecture relation and a complement. The complement defines one or more architecture concepts – that can be connected by coordinators – with which the subject is allowed to be related with or is not allowed to be related with. A sentence, i.e, an architecture rule, is finished with a full stop in order to indicate that the rule definition is completed.

Example 6.2.1: ArchCNL Sentence

An exemplary architecture rule in *ArchCNL* can be written as follows:

`Every` *Repository* `must` *manage* `an` *Entity* `.`

The exemplary *ArchCNL* sentence introduces two architecture concepts, namely *Repository* and *Entity*. Those two concepts are connected via the architecture relation *manage*.

`Every` *Repository* constitutes the subject concept expression. The keyword *Every* introduces the subject concept. *Repository* is the name of the concept for which the rule is defined. The verb phrase *manage an Entity* is the object concept expression. The concept *Entity* is called the *complement concept*. Both expressions use the simplest form of expressing concepts,

```
⟨specification⟩ ::= ⟨sentence⟩+

⟨sentence⟩ ::= (⟨subject⟩ ['must'|'can'] ⟨roleExpression⟩ ⟨object⟩'.')
  | ('If' ⟨conceptID⟩ ⟨roleName⟩ ('a'|'an') ⟨object⟩ ['then' | ','] 'it' 'must' ⟨roleName⟩
    'this' ⟨object⟩'.')

⟨subject⟩ ::= 'No' ⟨object⟩ | ('Every')? ⟨object⟩ | 'Everything' | 'Nothing'

⟨roleExpression⟩ ::= 'only' ⟨roleName⟩ ('a'|'an') | 'be' ('a'|'an') | ⟨roleName⟩
    ('at-most'|'at-least'|'exactly') ⟨count⟩ | ⟨roleName⟩ ('a'|'an')

⟨object⟩ ::= ⟨conceptName⟩ (⟨relativeClause⟩)? (('and'|'or') ⟨relativeClause⟩)*

⟨relativeClause⟩ ::= 'that' '(' ⟨roleName⟩ ⟨object⟩ ')'

⟨conceptName⟩ ::= ('A'..'Z')+(('A'..'Z')|('a'..'z'))*

⟨roleName⟩ ::= ('a'..'z')+('-'('a'..'z')+)*

⟨count⟩ ::= ('1'..'9')+
```

Grammar 6.1.: *Grammar of ArchCNL in EBNF.*

where the subject and object are *atomic concepts*. Atomic concepts are designated by *concept identifiers*. In subject and object expressions, relative clauses and coordinators can be used for a more comprehensive definition of subject and object concepts. In this case, subject and object concepts are *complex concepts*. These constructs will be elaborated in the next sections.

It is important to note that the concepts *Repository* and *Entity*, and the relation *manage* are not predefined by *ArchCNL*. The user of *ArchCNL* is free to choose the vocabulary for concepts and relations and to define their concrete meaning by writing *ArchCNL* sentences. Since ontologies are not restricted to specific concepts and relations, e.g., *component* or *module*, the architect can define architecture concepts and relations as they are needed for the project. This makes up the flexibility of the approach.

6.2.3. Vocabulary of ArchCNL

The vocabulary of *ArchCNL* consists of *content words* and a small set of *predefined keywords*. Content words are *concept identifiers* and *relation identifiers*:

- Concept identifiers are represented by a noun in singular form or plural in case of being part of a cardinality restriction[1]. A concept identifier is written as a word sequence in camel case.
- Relation identifiers are represented by verbs. Verbs are written in third person singular, in simple present tense, e.g., *accesses/access*, or in simple present tense in passive form,

[1]When written in a plural form, concept names are reduced to their word stem. For example, the concept *Repositories* is reduced to *Repository*. Stemming is implemented using the Standford NLP libraries [MSB$^+$14].

e.g, *is accessed by*[2]. The relation name or property name must start with a lower-case character. The relation name can be written as a sequence of words separated by hyphens.

Predefined keywords support the grammar of the language and are reserved for a special purpose. In contrast to content words, keywords are fixed and cannot be changed by the software architect or developer who formulates architecture rules. Keywords are independent of the application domain. The purpose of keywords is to form sentence structure by establishing relationships in a sentence. Keywords can be classified with the following categories:

- *Specifiers* are used to introduce an architecture concept. *ArchCNL* defines the specifiers `a`, `an`, `Every`, `No`, and `the`. The specifiers `a` and `an` are used depending on whether a noun starts with a consonant sound or a vowel sound.
- *Coordinators* combine verb phrases by the keywords `and` and `or`.
- *Subordinators* connect two sentences with each other. *ArchCNL* defines the keywords `if` and `then` to define conditional sentences.
- *Modal words* express modality. They are necessary to define the rule semantics of a sentence. *ArchCNL* defines the modal words `must`, `can`, and its derivations `only can`, and `can only`. Modal words are used to express permissions, obligations, and prohibitions in rule formalizations.

Some keywords are adopted from existing CNLs, e. g., [Kuh10], [DHK+07], that implement natural language layers for ontology languages and description languages. For example, the indefinite determiner `Every` of the specifier category is adopted to introduce a sentence.

6.3. Translating ArchCNL Sentences to Class Axioms

The meaning of *ArchCNL* constructs and sentence patterns is defined by *translational semantics* [Kle08]. According to [Kle08] the meaning of a language is specified by *"translating the program into another language that is well understood"*. In this context, *ArchCNL* sentences ("the program") are transformed to class axioms in description logics ("another language") which then define the meaning of the respective sentence.

The translations are considered *model transformations* [KWWB03] based on transformation rules. In the following, the transformation rules are presented. These transformation rules are applied for each *ArchCNL* sentence structure specified in order to obtain a semantically equivalent class axiom.

6.3.1. Mapping of Concepts and Relations

Concept identifiers – denoting atomic concepts – directly map to concept names in description logics. Relation identifiers map to role names. Complex concepts in a subject or object concept expression contain relative clauses, coordinators, or variables. In the subsequent paragraphs, the translation of those constructs to the corresponding constructs of description logics is described. In the following explanations, C, C_1, C_2, ... C_n, and D refer to concept expressions.

[2]Similar to concept names, verbs are reduced to their word stem using the stemming algorithms of the Stanford NLP libraries [MSB+14]. For example, the relation *accesses* is reduced to its word stem *access*.

A concept expression is either an atomic concept or a complex concept. The explanations refer to the term *concept* to indicate both. The literals R and S denote relation identifiers. Keywords in *ArchCNL* sentences are emphasized in `typewriter font`.

Relative Clauses

Concept expressions defined in architecture rules can be arbitrarily complex by using relative clauses. Relative clauses are used to modify a noun, i.e., a subject or an object architecture concept, in an *ArchCNL* sentence. A relative clause is introduced by the `that` keyword. It modifies the immediately preceding architecture concept. The restriction relates the concept to be modified with another concept, variable, or data value (string or number) by an architecture relation.

A relative clause follows the sentence structure

$$C\ \texttt{that (}R\ \texttt{a/an}\ D\texttt{)}$$

where C is the concept that is modified by the relative clause `(`R `a/an` D`)`. The parentheses are used for increasing the clarity of the formalization. The relative clause only modifies the immediately preceding architecture concept in order to eliminate ambiguity. A relative clause is introduced by the `that` keyword directly after the architecture concept that should be modified by the relative clause. D is the *modifying concept* that modifies C. C is the *modified concept.* The keyword `that` is mapped to the intersection in the description logics formalism. Consequently, a relative clause is transformed to description logics as follows

$$C \sqcap R.D$$

For example, an architecture rule using relative clauses could be expressed as follows:

`Every` *Entity* `that (`*is-located-in* `an` *EntityPackage*`) can only` *use* `an` *Entity* `that (`*is-located-in* `an` *EntityPackage*`).`

which is transformed to

$$\mathit{Entity} \sqcap \mathit{is\text{-}located\text{-}in}.\mathit{EntityPackage} \sqsubseteq \forall \mathit{use}.(\mathit{Entity} \sqcap \mathit{is\text{-}located\text{-}in}.\mathit{EntityPackage})$$

In this rule, the noun *Entity* is modified by the relative clause *is-located-in* `an` *EntityPackage* (in the subject and object expression). *is-located-in* is the modifying relation, whereas *EntityPackage* is the modifying concept in the relative clause. This means, that the rule does only apply to instances of the concept *Entity* in a specific package (*EntityPackage*), and not on all instances of the concept *Entity* in the entire software system.

The syntax of relative clauses is defined recursively. Consequently, a relative clause can be arbitrarily complex by nesting relative clauses. In this way, the modifying concept can also be modified by a relative clause. However, in favor of understandability, the nesting of relative clauses should be well considered.

Coordinators

Verb phrases can be combined by using the coordinators `and` and `or` in order to allow for a comprehensive formalization of architecture rules. In this way, a combination of concepts can be used as a range of a restriction. The coordinator `and` is mapped to intersection ($\sqcap$) of concept descriptions, while the coordinator `or` is mapped to the union ($\sqcup$) of concept descriptions. Verb phrases that are connected by `and` are transformed to existential restrictions connected by intersection. This means the structure

$$R \texttt{ a/an } C_1 \texttt{ and } R \texttt{ a/an } C_2 \ldots \texttt{ and } R \texttt{ a/an } C_n$$

is transformed to

$$\exists R.C_1 \sqcap \exists R.C_2 \ldots \sqcap \exists R.C_n$$

The transformation for `or`-coordination is performed accordingly. Coordination of verb phrases can be used in relative clauses or in the object concept expression.

For example, the following rule uses `and` coordination in the object concept expression:

`Every` *Entity* `must` *provide* `an` *IDField* `and` *provide* `a` *NameField*`.`

In this example, individuals of the concept Entity must establish both a *provide*-relation to individuals of *IDField* and *NameField*. This sentence is mapped to the corresponding axiom

$$\mathit{Entity} \sqsubseteq \mathit{provide}.\mathit{IDField} \sqcap \mathit{provide}.\mathit{NameField}$$

Variables

Variables allow to differentiate between concrete instances of the same concept. For example, the rule

$$\mathit{Entity} \sqcap \mathit{is\text{-}located\text{-}in}.\mathit{EntityPackage} \sqsubseteq \forall \mathit{use}.(\mathit{Entity} \sqcap \mathit{is\text{-}located\text{-}in}.\mathit{EntityPackage})$$

does not differentiate between different instances of *EntityPackage*s. Assuming that entities are only allowed to access other entities in the same entity package, a variable X can be introduced to the rule as follows:

`Every` *Entity* `that (`*is-located-in EntityPackage X*`) can only` *use* `an` *Entity* `that (`*is-located-in EntityPackage X*`).`

Variables are mapped to concrete instances of a knowledge base. In this example, the phrase *is-located-in EntityPackage X* is mapped to the corresponding construct in description logics

$$\mathit{is\text{-}located\text{-}in}.\{X\}$$

where X stands for any individual of the concept *EntityPackage*. For each individual of *EntityPackage* that can be found in the knowledge base, a corresponding axiom is created that substitutes the variable with a concrete individual of the knowledge base. E.g., let two individuals *entityPackage*1 and *entityPackage*2 belonging to the concept *EntityPackage* be stored in the knowledge base, then the following two axioms are created:

$$\mathit{Entity} \sqcap \mathit{is\text{-}located\text{-}in}.\{\mathit{entityPackage}1\} \sqsubseteq \forall \mathit{use}.(\mathit{Entity} \sqcap \mathit{is\text{-}located\text{-}in}.\{\mathit{entityPackage}1\})$$

$$\mathit{Entity} \sqcap \mathit{is\text{-}located\text{-}in}.\{\mathit{entityPackage}2\} \sqsubseteq \forall \mathit{use}.(\mathit{Entity} \sqcap \mathit{is\text{-}located\text{-}in}.\{\mathit{entityPackage}2\})$$

Differently named variables correspond to different individuals. For example, in the previous example, the individuals *entityPackage*1 and *entityPackage*2 are different, i.e., $\mathit{entityPackage}1 \not\approx \mathit{entityPackage}2$.

Table 6.1.: *Supported rule types of the formalism with exemplary rules in description logics and in ArchCNL. C and D are atomic concepts or concept descriptions, R and S are atomic roles, n is a natural number.*

Rule Type	Description Logic	ArchCNL expression	Example in ArchCNL
Is-A Rule	$C \sqsubseteq D$	`Every` C `must be a` D.	`Every` *AggregateRoot* `must be an` *Entity*.
Only-Can Rule	$\exists R.\top \sqsubseteq C$, $\top \sqsubseteq \forall R.D$	`Only a/an` C `can` R `a/an` D.	`Only a` *ServiceComponent* `can` *use* `a` *DAO*.
Must Rule	$C \sqsubseteq \exists R.D$	`Every` C `must` R `a/an` D.	`Every` *Repository* `must` *manage* `an` *Entity*.
Can-Only Rule	$C \sqsubseteq \forall R.D$	`Every` C `can only` R `a/an` D	`Every` *LogicType* `can only` *access* `a` *StorageApi*.
Cardinality Rule	$C \sqsubseteq = nR.D$ $C \sqsubseteq \leq nR.D$ $C \sqsubseteq \geq nR.D$	`Every` C `can` R `exactly/at-most/at-least n` D.	`Every` *Host* `can` *contain* `(exactly, at most, at least) 2` *ServiceInstances*.
If-Then Rule	$R \sqsubseteq S$ $\exists R.\top \sqsubseteq C$, $\top \sqsubseteq \forall R.D$ $\exists S.\top \sqsubseteq C$, $\top \sqsubseteq \forall S.D$	`If` C R `a/an` D, `then it must` S `this` D.	`If a` *LogicType* *uses* `a` *DBType*`, it must` *manage* `this` *DBType*.
Negation Rule	$C \sqsubseteq \neg(\exists R.D)$ $C \sqsubseteq \neg(\exists R.\top)$ $\top \sqsubseteq \neg(\exists R.D)$	`No` C `can` R `a/an` D. `No` C `can` R `anything.` `Nothing can` R `a/an` D.	`No` *DAO* `can` *use* `a` *BusinessLogicComponent*.

6.3.2. Mapping Sentence Structures to Rule Types

The rule types are described in terms of sentence structures. These structures prescribe how an architectural rule type is allowed to be written. Having such a well-defined syntax for the rule types allows for a systematic mapping between sentences and class axioms.

In the following, the seven types of architecture rules that are fundamental for supporting architecture rule formalization are proposed. Note that the given rule types can be easily extended with new ones, as description logics support a flexible definition of concepts by using concept constructors. Table 6.1 depicts the rule types in *ArchCNL* and their formal representation in description logics and an example for each rule type. In the following, the rule types and their corresponding representation in *ArchCNL* are explained in more detail.

Is-A rule Type

The rule type describes that an individual of an architecture concept must also satisfy the properties of its parent architecture concept. For this rule type, the sentence structure

`Every` C `must be a/an` D.

is defined, where C is the subject concept that is restricted by the concept D. This rule type maps to a expression in description logics of the form

$$C \sqsubseteq D$$

Basically, the rule type expresses a hierarchical relationship between C and D (C is the subconcept of D) due to the $\sqsubseteq$ relation. That is why, the keywords `must be a` naturally map to this relation. This rule is violated when an individual has been classified as concept C, but not as concept D.

Only-Can Rule Type

This rule type expresses than an individual has to be of a specific concept in order to be allowed to have a specific relation to another individual of another concept. The rule type follows the sentence structure

`Only a/an` C `can` R `a/an` D`.`

where C and D are concepts and R is an relation. The rule is violated when an individual of another concept different than C can be found that establishes the specified relationship R with an individual of concept D. This rule type is used to detect *divergences*. It maps to one domain and one range restriction. For the relation R the concept C is defined as its domain, whereas the concept D is defined as its range. Consequently, this rule type maps to the axioms

$$\exists R.\top \sqsubseteq C, \tag{6.1}$$
$$\top \sqsubseteq \forall R.D \tag{6.2}$$

Axiom 6.1 defines the domain restriction, whereas axiom 6.2 defines the range restriction.

Must Rule Type

Every individual of a specific concept must have a specific relation with an individual of another architecture concept. This rule type is written as

`Every` C `must` R `a/an` D`.`

The rule is violated when an individual of concept C misses to establish the relation R with an individual of concept D. The rule type is used to reveal *absences*.

This sentence structure is a verbalization of the existential restriction of the description logics formalism of the form:

$$C \sqsubseteq \exists R.D$$

Can-Only Rule Type

Every individual of a specific concept can only have a specific relation with an individual of a specific architecture concept. For this rule type the following sentence structure is defined:

`Every` C `can only` R `a/an` D`.`

In case an individual of concept C establishes the relation R with an individual of a concept different from D, the rule is violated. The rule type reveals *divergences*.

The meaning of `can only` maps to the universal restriction. That is why, the sentence structure can be mapped to the following axiom

$$C \sqsubseteq \forall R.D$$

where C and D are the concepts and R is the relation name as used in the sentence structure.

Cardinality Rule type

Every individual of a specific architecture concept must have at most/at least/exactly n relations to an individual of another architecture concept, where n is a natural number, written as

`Every` C `must` R `exactly/at-most/at-least` n D.

This rule type is based on the cardinality restriction of the description logics formalism:

$$C \sqsubseteq = nR.D$$
$$C \sqsubseteq \leq nR.D$$
$$C \sqsubseteq \geq nR.D$$

As in the previous rule types, C and D refer to the concepts used in the sentence structure and R refers to the relation name. In addition, a natural number n is defined in order to specify the number of relations that must exist between the individuals of both concepts.

If-Then Rule type

This rule type expresses an obligation. An individual of a specific concept C that is related with another individual of concept D via the relation R must also be related with the individual of concept D via the relation S. This is expressed as a conditional sentence using the subordinators `if` and `then`:

`If` C R `a/an` D, `then` C `must` S D.

As opposed to the previous rule types, two relations must be specified. These are denoted as R and S in the sentence structure above. This rule type distinguishes from the other rule types as it describes a role inclusion axiom, i.e., a relationship between two roles and not between two concepts:

$$R \sqsubseteq S$$

R and S refer to the relations as used in the sentence structure above. Note that the concepts C and D are not explicitly captured in this axiom. That is why, the corresponding domain and range restrictions must be defined for the two roles. From the sentence it can be derived that C is the domain of R and S and that D is the range of R and S:

$$\exists R.\top \sqsubseteq C,$$
$$\top \sqsubseteq \forall R.D$$
$$\exists S.\top \sqsubseteq C,$$
$$\top \sqsubseteq \forall S.D$$

The rule is violated when an individual of concept C establishes a relation R with an individual of concept D, but misses to establish the relation S with the individual of D. This rule type is necessary in order to detect *absences*.

Negation Rule type

This rule is necessary in order to prohibit an individual of a concept C to be related with another individual of another concept D. The syntactic structure of the negation rule type is as follows:

`No` C `can` R `a/an` D`.`

The rule type is formalized in description logics using the negation constructor:

$$C \sqsubseteq \neg(\exists R.D)$$

A violation of this rule type occurs when an individual of concept C establishes a relation R with an individual of concept D.

Additionally, it is possible to express that no individual of any architecture concept is allowed to establish an architecture relation with a given architecture concept. Instead of listing all architecture concepts defined for the project, the rule can be shortened to

`Nothing can` R `a/an` D`.`

The keyword `Nothing` is introduced to *ArchCNL* in order to express this rule type. It can only be used as a subject concept. This sub-type is formalized as

$$\top \sqsubseteq \neg(\exists R.D)$$

Intuitively, this formalization states that every concept (designated by the top concept $\top$) is not allowed to establish a relation R with the architecture concept D. Consequently, this also means, that individuals of D are not allowed to be related with other individuals of D. E.g., "`Nothing can` *access* `a` *Controller*" means that an individual belonging to the concept *Controller* is not allowed to access another individual that belongs to the concept *Controller*[3].

Accordingly, it is possible to express that a specific architecture concept is not allowed to be related with any other concepts by a relation of a specific type. This is specified as:

`No` C `can` R `anything.`

The keyword `anything` is introduced in order to simplify the rule specification. Instead of listing all possible concepts, the keyword can be used as an abbreviation. It can be used only as an object concept. The sentence structure maps to the following axiom:

$$C \sqsubseteq \neg(\exists R.\top)$$

[3]Please note that if this semantics is not intended the *can-only* rule type can be applied, i.e., `Only a Controller can access a Controller`

6.4. Transformation Algorithm

From the *ArchCNL* sentences written by the software architect, an OWL ontology is generated automatically. This ontology stores architecture concepts as OWL classes, architecture relations as object properties or data type properties, and *ArchCNL* sentences, i.e., architecture rules, as class axioms. The resulting OWL ontology is called the *architecture ontology*:

Definition 6.4.1: Architecture Ontology

An architecture ontology is the result of transforming *ArchCNL* sentences into an OWL ontology representing the architecture concept language.

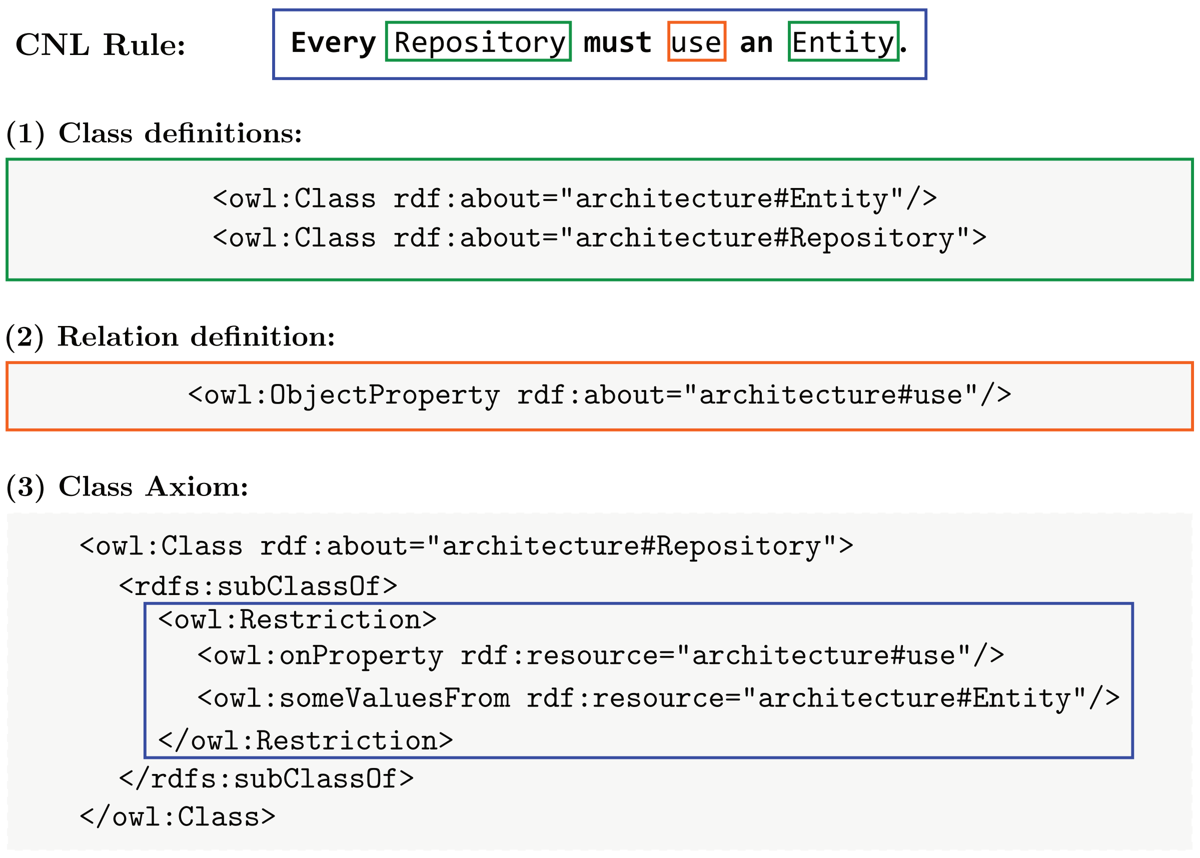
The ontology is stored as an OWL file. The grammar of the *ArchCNL* and the transformation from *ArchCNL* to OWL are implemented using Xtext [xteb] technologies.

It is important to note that no additional statements for defining the architecture concepts and relations are necessary. While writing architecture rules as *ArchCNL* sentences, the architecture concepts and relations are automatically extracted from the sentences and stored as OWL classes and properties (i.e., concepts and roles in description logics terms). For example, when writing `Every` *Repository* `must` *use* `an` *Entity*, the classes *Repository* and *Entity*, and the property *use* are stored in a OWL ontology without explicitly stating that they are architecture concepts and relations. Furthermore, the *ArchCNL* sentence is transformed into a class axiom using the extracted concepts and relations.

In general, the following steps are performed during the transformation:

1. The ontology that stores the architecture concepts, relations, and rules as OWL classes, properties, and axioms is initialized.
2. Each sentence is checked for syntax errors. If the sentence is not a valid according to the grammar (see Grammar 6.1), the algorithm terminates with an error.
3. For each sentence, the rule type is identified and the corresponding transformation to description logics according to the mapping as described in the previous sections is chosen.
4. The subject concept expression is transformed. In this step, the architecture concepts and relations specified in the expression are identified. For each of them, OWL classes and properties are generated.
5. According to the previous step, the object concept expression is transformed depending on the identified rule type.
6. Finally, subject and object concept expressions are connected via a subclass axiom and the rule is added to the ontology.

Figure 6.1 shows an excerpt of an OWL ontology resulting from transforming the *ArchCNL* sentence `Every` *Repository* `must` *use* `an` *Entity* according to the steps described before. The ontology contains two class definitions (Figure 6.1 **(1)**). These class definitions correspond to the concept identifiers used in the *ArchCNL* sentence. Additionally, the ontology defines an object property named `use` representing the architecture relation used in the *ArchCNL* sentence (Figure 6.1 **(2)**). The actual formalization of the rule is depicted in Figure 6.1 **(3)**. It



CNL Rule: `Every Repository must use an Entity.`

(1) Class definitions:

```
<owl:Class rdf:about="architecture#Entity"/>
<owl:Class rdf:about="architecture#Repository">
```

(2) Relation definition:

```
<owl:ObjectProperty rdf:about="architecture#use"/>
```

(3) Class Axiom:

```
<owl:Class rdf:about="architecture#Repository">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="architecture#use"/>
      <owl:someValuesFrom rdf:resource="architecture#Entity"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Figure 6.1.: *Result of an exemplary transformation. The prefix* `architecture` *is used as an abbreviation for the URI of the architecture ontology.*

defines an OWL restriction on the relation `use` using the `someValuesFrom` expression. This corresponds to the existential restriction (∃) in description logics. As defined in the mapping rules in the previous sections, the *Must* rule type is mapped to this type of restriction.

6.5. Conclusion

In this chapter, *ArchCNL* for specifying the architecture concept language based on architecture rules written as natural language sentences is presented. This specification is automatically transformed to an OWL ontology. This is necessary in order to enable automatic analyses of the architecture rules specified in *ArchCNL*. As a next step, the ontology containing the specified architecture concepts, relations, and rules – represented as class axioms – is used to verify whether the implementation follows the architecture concept language, i.e., to check the conformance of the implementation against the architecture rules. This step is elaborated in the following chapter (Chapter 7).

7. ArchCNLCheck: Ontology-Based Architecture Conformance Checking

In the previous chapter, the first phase of the approach (compare Figure 5.2) is described, i.e., how an architecture concept language and architecture rules are formalized by writing natural language sentences with the *Architecture Controlled Natural Language (ArchCNL).* Additionally, it is described how these sentences are transformed into an ontology representing the language and the architecture rules.

In this chapter, *ArchCNLCheck* is presented which implements the second phase of the approach (compare Figure 5.2), i.e., *ontology-based architecture conformance checking.* It is ontology-based since every single information necessary for performing this process is represented as and stored in an ontology. Additionally, description logics reasoning is applied to implement conformance checking. *ArchCNLCheck* implements the following steps: 1) representing source code artifacts as ontologies, 2) applying architecture-to-code-mapping on the ontology-based source code artifact representation using mapping rules, 3) applying reasoners for computing architecture violations, and 4) storing conformance checking results as ontologies so that they can be retrieved afterwards.

The main idea *ArchCNLCheck* is to exploit reasoning services of the description logics formalism (see Section 2.4.4) in order to implement architecture conformance checking. The detection of architecture violations is considered as the activity of checking for inconsistencies in the implemented architecture with respect to architecture rules. Together, the architecture rules and the implemented architecture represent a knowledge base. Intuitively, the implemented architecture is the ABox of the knowledge base, whereas the architecture concept language (specifying architecture rules) represents the TBox of the knowledge base. Thus, architecture conformance checking can be reduced to the task of reasoning for the consistency of the ABox, i.e., the implemented architecture, with respect to the TBox, i.e., the architecture concept language and architecture rules. This means, the knowledge base is investigated for inconsistencies. The reasoner returns *true* if the ABox is consistent with regard to the TBox., i.e., the implemented architecture conforms to the architecture rules. Otherwise, the reasoner returns *false* and the implemented architecture violates the rules. However, the consistency reasoning service alone does not reveal the concrete architecture violations in the implemented architecture. That is why, reasoners additionally allow to list explanations helping to find the exact reasons for inconsistencies. An explanation contains a set of axioms that caused the inconsistencies [HPS10]. These explanations are used to extract which violations occurred and why architecture rules are violated.

In the following sections, ontology-based conformance checking process implemented by *ArchCNLCheck* is described in more detail. Section 7.1 defines relevant terms of the process and how the steps of conformance checking are conducted in general. Section 7.2 presents the ontologies for representing source code. In Section 7.3 the architecture-to-code-mapping by SWRL rules is explained. Finally, the actual conformance checking step is presented in Section 7.4, i.e., how architecture violations are calculated based on the output of the reasoner and how the results are preserved as ontologies (Section 7.5). Section 7.6 presents the tool chain that automates the conformance checking process.

7.1. Ontology-Based Architecture Conformance Checking: Terms and General Process

In this section, the ontology-based architecture conformance checking process implemented by *ArchCNLCheck* is described. Additionally, the meaning of terms important to conformance checking, e.g., implemented architecture or architecture violation, are redefined for the context of ontology-based architecture conformance checking.

In order to use reasoning services for architecture conformance checking, several steps are necessary. The overall conformance checking process is depicted in Figure 7.1. Each step will be explained briefly in the following. A detailed explanation of each step is given in the respective section. To perform architecture conformance checking, the following inputs are necessary:

1. the architecture concept language that encompasses architecture concepts, relations, and architecture rules specified in *ArchCNL*,

2. the source code artifacts that will be checked for conformance (e.g., Java source code), and

3. an architecture-to-code-mapping in order to identify architecture concepts and relations in the source code.

The architecture rules are provided by the software architect who writes *ArchCNL* sentences (Chapter 6) in order to formalize the rules. As described in the previous chapter, these sentences are automatically transformed into corresponding class axioms and are stored as an OWL ontology (step CNL2OWL in Figure 7.1). This ontology is denoted as architecture ontology (Definition 6.4.1). The *ArchCNL* and the transformation of sentences into an OWL ontology have already been described in Chapter 6. Furthermore, in this chapter, an architecture rule is defined more precisely as:

Definition 7.1.1: Architecture Rule

An architecture rule $r \in R$ (where R is the set of architecture rules of the architecture concept language as defined in Definition 5.1.4) is a triple $r = (cnl, type, id)$ where

- cnl is the *ArchCNL* representation of the rule,

- $type \in \{is-a, must, can-only, only-can...\}$ is the type of the rule (see Chapter 6), and

- $id \in \mathbb{N}$ is the ID of the rule (created during transformation)

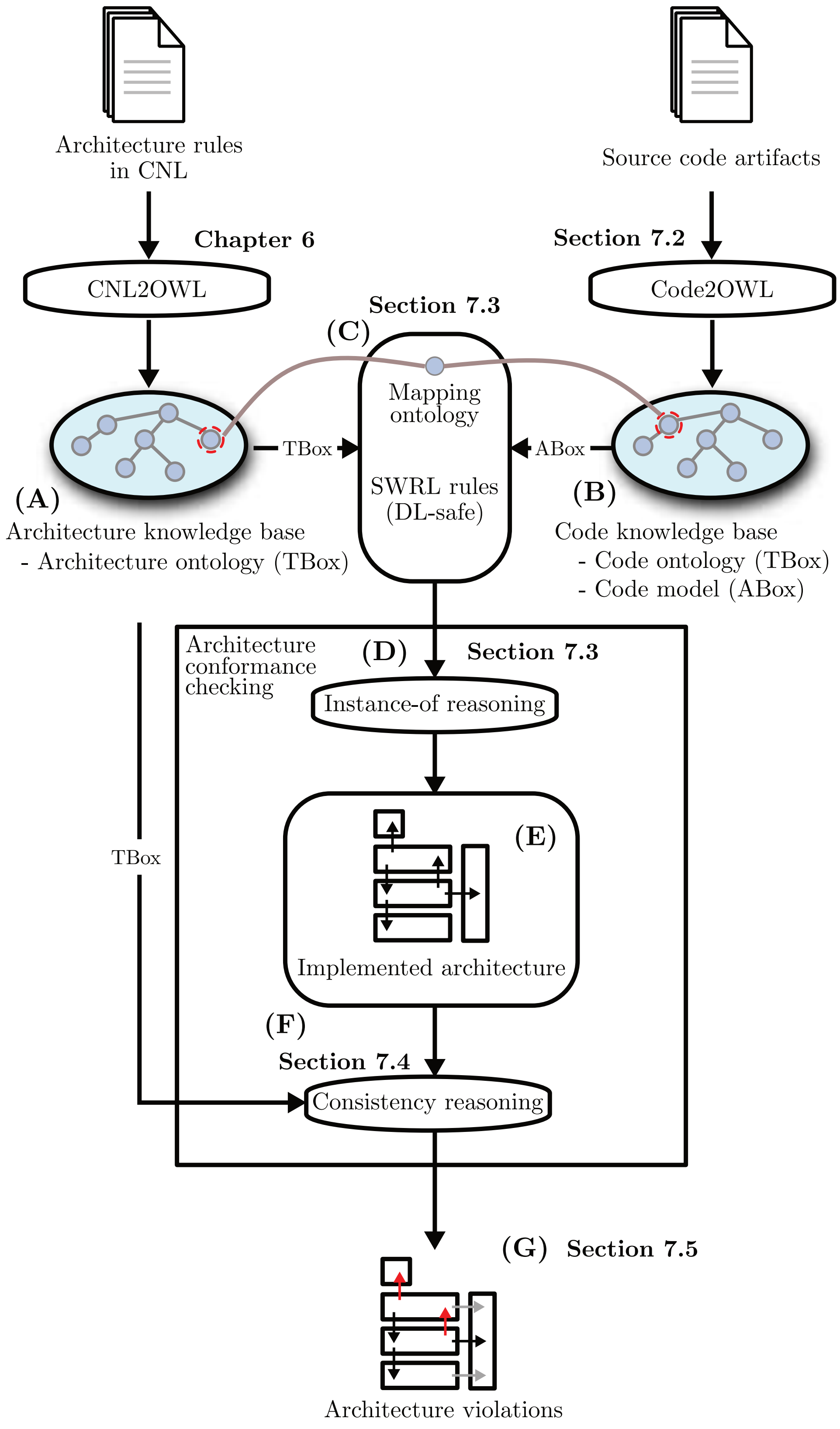


Figure 7.1.: *The conformance checking process.*

The architecture ontology is validated against the implementation in a next step. For this, the implementation consisting of several source code artifacts must be transformed to an ontology-based representation. A source code artifact is defined as follows:

Definition 7.1.2: Source Code Artifact

A source code artifact is a piece of information that is produced, modified, or used during the architectural implementation phase.

A source code artifact can be categorized according to a type, called *artifact type* [MFC+18], e.g., Java code, deployment descriptor, Maven build configuration, Git history etc. The main characteristics of an artifact type can be captured in an ontology in terms of concepts and relations. In the context of this thesis, this is called *code ontology*.

Definition 7.1.3: Code Ontology

A code ontology is an ontology that represents the domain of a source code artifact type. A code ontology consists of source code concepts - represented by description logics concepts – and source code relations – represented by roles of the description logics formalism. A code concept represents a type of a code element specific to an artifact type, whereas a code relation connects two code concepts.

The code ontology does not contain architecture concepts and relations. The concepts and relations defined in the code ontology are project-generic in contrast to the architecture ontology. Once defined, the code ontology can be reused across different projects. Based on a code ontology, a source code artifact can be described as an ontology-based model, called *code model*:

Definition 7.1.4: Code Model

A code model is an ontology-based representation of source code artifacts. It contains individuals of one or more code ontologies.

Together, the code ontology and the code model constitute the *code knowledge base*. The code ontology is considered the TBox, whereas the code model is considered the ABox of the code knowledge base.

In this thesis, three ontologies for source code artifacts are designed, namely for object-oriented software systems based on the FAMIX meta model, for Maven artifacts, and for Git repositories. The code model is automatically created from source code artifacts (step **(B)** in Figure 7.1). *ArchCNLCheck* provides parsers to transform source code artifacts to their corresponding ontology-based representation. In Section 7.2, the ontologies are presented.

As described in Section 2.3.2, the implemented architecture needs to be extracted from the source code, i.e., the source code is lifted to an architectural level. For this, a mapping between architecture and code needs to be described. The mapping is defined in a so-called *mapping ontology* provided by the software architect (step **(C)** in Figure 7.1):

Definition 7.1.5: Mapping Ontology

A mapping ontology is an ontology defined based on combining the architecture ontology, the code ontology, and the code model. It connects the concepts and relations of the architecture ontology with the concepts and relations of the code ontology and the individuals of the code model for which the conformance will be checked. Furthermore, it defines a collection of *mapping rules*.

Definition 7.1.6: Mapping Rule

A mapping rule describes how the concepts and relations of a code ontology relate to concepts and relations of the architecture ontology. The mapping rule describes different properties individuals and relations of the code model must satisfy in order to be classified as an architecture concept or relation.

Mapping rules rely on different kinds of information from the source code artifacts. This can be, beside others, the package structure, names of source code entities, or meta data information (e.g. annotations). The mapping rules are formalized in SWRL [swr04] (Section 2.4.7). These rules are applied by invoking reasoning services. The *instance-of* relation is calculated for the individuals of the code model (step **(D)** in Figure 7.1). As a result, code elements are represented in terms of architecture concepts and relations, i.e., those concepts and relations that are defined in the architecture ontology. This means, the *implemented architecture* is extracted from the code (step **(E)** in Figure 7.1):

Definition 7.1.7: Implemented Architecture in an Ontology-Based Sense

The implemented architecture IA is the code model containing individuals enriched with architecture concepts and relations from the architecture ontology.

In a next step, the implemented architecture is then checked against the axioms defined in the architecture ontology using the consistency reasoning service **(F)** in order to reveal architecture violations **(G)**:

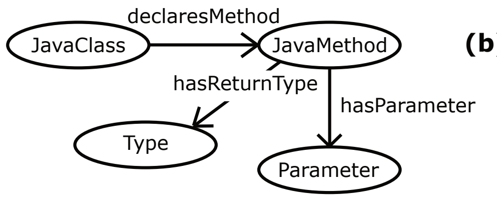
Definition 7.1.8: Architecture Violation in an Ontology-Based Sense

A violation of an architecture rule is an inconsistency in the implemented architecture with respect to the architecture ontology detected by the description logics reasoner.

7.2. Ontology-Based Representation of Source Code Artifacts

The approach requires all necessary artifacts to be represented uniformly as ontologies, so that architecture rules described as class axioms can be validated on the source code level. In this thesis, an extensible set of ontologies is designed that allows for the description of source code artifacts as individuals based on those ontologies. Figure 7.2 shows how such an ontology should minimally look like in order to represent the given code snippet written in Java. The

(a)



(b)

```
class PersonRepository {
  public Person getPersonByName(String name) {
            ...
  }
}
```

(c)

```
JavaClass(PersonRepository), JavaClass(Person), JavaMethod(getPersonByName),
declaresMethod(PersonRepository, getPersonByName), hasReturnType(getPersonByName,Person),
hasModifier(getPersonByName,"public"), Parameter(name), hasParameter(getPersonByName,name)
PrimitiveType(String), hasDeclaredType(name,String)
```

Figure 7.2.: *Representing software structures based on ontologies. (a) An example for a Java code ontology modeling concepts for classes, methods, parameters, and primitive types. (b) A code snippet of a Java class. (c) Facts (concepts and role assertions) representing the code snippet from (b).*

ontology - illustrated as a graph in Figure 7.2 (a) - should contain concepts such as *JavaClass*, *JavaMethod*, or *Parameter* to define the structure of the code snippet in form of constructs that are characteristic for Java code. Additionally, it defines relations between the concepts in order to describe code-level relationships between Java code constructs. For example, the ontology defines the relation *declaresMethod* to denote that a Java class defines a specific method. The exemplary Java code in Figure 7.2 (b) can then be represented as individuals of the code ontology.

Using ontologies, heterogeneous types of source code artifacts can be integrated into one single representation. Different types of source code artifacts can be integrated with each other by explicitly describing relationships between them. This allows for more comprehensive architecture analyses compared to existing approaches that only allow for analyses on one type of source code artifact, e. g., Java code.

For each code ontology, a fact extractor is provided. Such a fact extractor parses the main constructs of the respective artifact and transforms them to the code model. For example, a Java fact extractor parses the *abstract syntax tree* (AST) that represents the abstract syntactic structure of a language construct, e.g., a Java class, in a first step. Then, the AST is transformed to individuals of the concept, e.g., the concept *JavaClass*, that represents the corresponding language construct.

The approach does not claim that the ontologies are completely specified. The main aim is to capture the main concepts and relations of each domain. As ontologies are easily extendable, concepts and relations can be added as needed. The ontologies have been designed following the processes and best practices of ontology design as described in [NM$^+$01].

In the following sections, ontologies for various software related artifacts like object-oriented source code, build files, and data from version control are presented. When describing the ontologies, concept names are denoted in CapitalLetters, whereas relations (object properties and data type properties) are underlined. The ontologies are stored as OWL ontologies. Each ontology is uniquely identified by a namespace URI. Each concept, relation, and axiom is referenced by its URI having the namespace as a prefix of the ontology it belongs to. Table 7.1 lists the ontologies designed in this thesis and their respective URI. When referencing a specific ontology and its elements, the prefix is used as an abbreviation of the namespace.

Table 7.1.: *Ontologies, their URI, and the corresponding prefix referencing the URI*[1]

Ontology	URI	Prefix
Source Code Artifact	http://arch-ont.org/ontologies/main.owl	main
FAMIX	http://arch-ont.org/ontologies/famix.owl	famix
Git	http://arch-ont.org/ontologies/git.owl	git
Maven	http://arch-ont.org/ontologies/maven.owl	maven

7.2.1. General Source Code Artifact Ontology

The *General Source Code Artifact Ontology* designed in this thesis allows for the integration of different artifact types. This ontology is not bound to a specific domain. This means that it defines concepts and relations that are omnipresent for all domains describing source code artifacts. Intuitively, it describes a top layer for all other ontologies which inherit the concepts and relations from this general ontology. This kind of ontology is a so-called *upper ontology*. An upper ontology describes *"the organization of the real-world knowledge in terms of the most general concepts"* [Rat09]. The main aim of this ontology is to capture all concepts and properties that are necessary to describe more specific types of source code artifacts, as Java source code, build files, version control data etc. Inheriting ontologies refine the meaning of the more abstract concepts and relations defined in the general ontology as it will be illustrated later for the Java, Maven, and Git ontologies. This general ontology is a result of a bottom-up design process. First, more specific ontologies for source code artifacts have been designed, e.g. Java source code ontology, maven ontology, Git ontology. Then, common concepts and relations have been identified between the existing ontologies that were then abstracted to this separate ontology.

The ontology is depicted in Figure 7.3. It defines three general concepts, namely ARTIFACTTHING, SOURCECODEARTIFACT, and SOURCECODEARTIFACTFILE. The root concept ARTIFACTTHING represents a so-called meta concept. The concepts SOURCECODEARTIFACT and SOURCECODEARTIFACTFILE are sub-concepts of this concept and therefore inherit the characteristics and attributes of this meta concept. An ARTIFACTTHING has a name (hasName) and is uniquely identified by an ID (hasID). Therefore, the sub-concepts SOURCECODEARTIFACT and SOURCECODEARTIFACTFILE are associated with a name and an ID. Additionally, individuals of the concept ARTIFACTTHING are connected with a time stamp indicating when an ARTIFACTTHING (i.e., SOURCECODEARTIFACT and SOURCECODEARTIFACTFILE) has been created. The concept SOURCECODEARTIFACT represents a product that is produced during the design and/or development of a software system. Different artifacts can depend on each other. A dependency between artifacts is modeled by the object property dependOn. This object property models a very generic relationship that can be established between arbitrary things in the context of the source code artifact domain. This relation generalizes different types of relations, e.g., dependencies between source code entities, relationships between documents (software architecture, requirements, source code comment etc). Such relations are refined in more specific ontologies in which relations inherit from this relation. For example, in the Java

[1]Please note that the URI is not a prescribed website where the ontology is stored. The URI can be defined arbitrarily and does not have to reference an existing website.

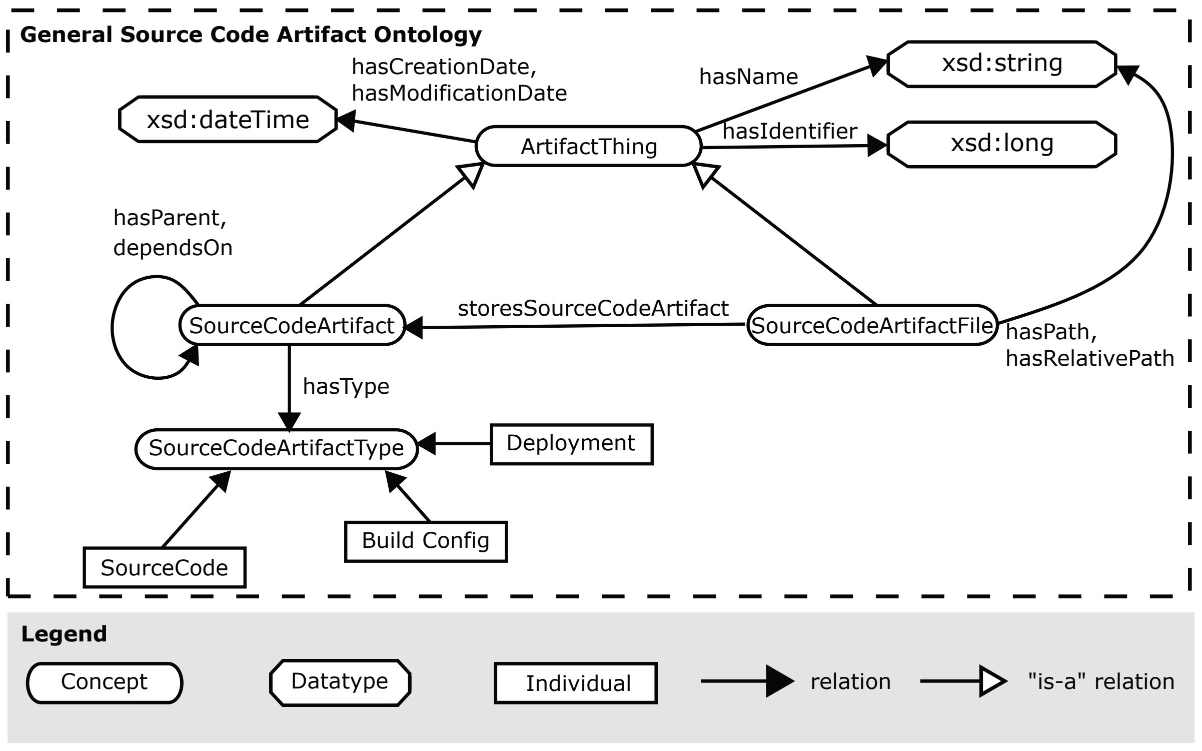


Figure 7.3.: *Overview on the general source code artifact ontology containing concepts and relations that are common for different source code artifact types.*

source code ontology a dependency can exist between two classes, where one class uses the other class as a type for a field declaration.

Artifacts can be organized in a hierarchy using the hasParent relation. Source code artifacts are persisted in files. Therefore, the SourceCodeArtifactFile concept represents a file on a filesystem that stores a source code artifact (storesSourceCodeArtifact). This is for example a Java file that stores a Java class definition or an XML file that stores a build configuration. A SourceCodeArtifactFile is stored on a specific location in a filesystem specified by a file path (hasPath and hasRelativePath).

7.2.2. Representing Object-Oriented Source Code as an Ontology

In order to design the ontology for object-oriented source code, the FAMIX meta model [DAB$^+$11] is used as a reference meta model in order to derive essential concepts and relations for the ontology. FAMIX provides a common meta model for object-oriented languages – e.g. Java and C# - in order to represent facts about the software under analysis in a language-independent manner. It aims to provide a standardized interexchange format for source code models. It is well-specified and therefore has a suitable level of formality. It is sufficiently detailed as it provides a comprehensive set of concepts to describe object-oriented source code. In the following, the main components of the FAMIX meta model are described, so that, in the next step, the ontology can be derived. *ArchCNLCheck* provides a parser that automatically transforms source code entities, such as classes, methods, or attributes, into individuals of the FAMIX ontology.

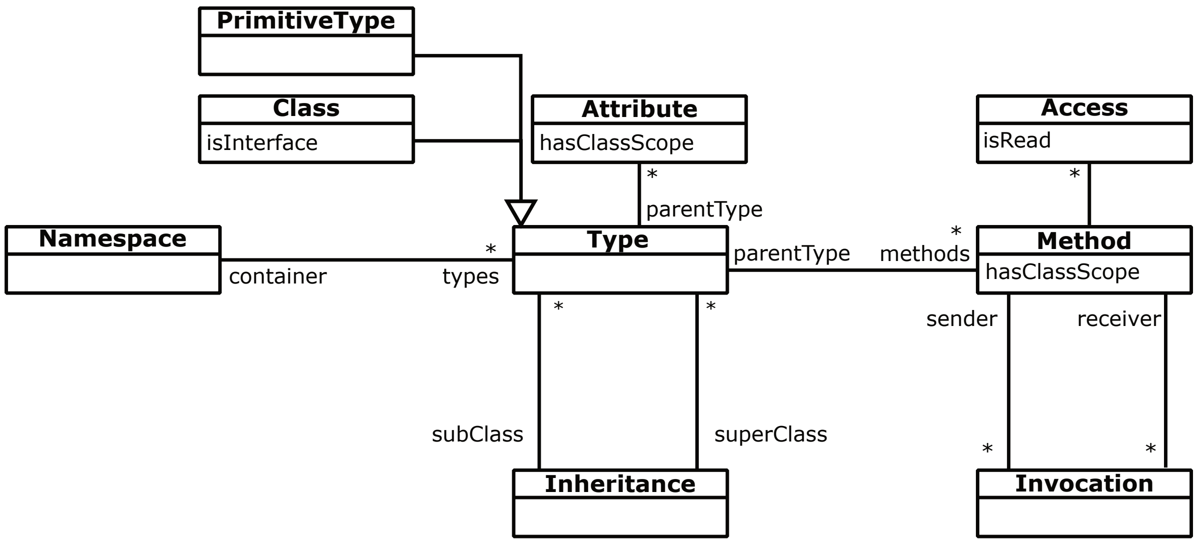


Figure 7.4.: *Excerpt of the FAMIX meta model showing the core elements.*

7.2.3. FAMIX Core Meta Model

In Figure 7.4 an excerpt of the core elements of the FAMIX meta model is depicted. In order to distinguish meta classes and ontology concepts from each other, meta classes are emphasized in **bold**. Associations and attributes of the FAMIX meta model are emphasized as <u>**bold underlined**</u> verbs. The FAMIX meta model describes its main components as a UML class diagram. The elements of the FAMIX meta model are described as UML classes. Relations between the elements are modeled as binary, bidirectional associations. The FAMIX meta model represents structural entities that are common to all object-oriented programming languages, such as classes, methods, variables, statements and expressions. The meta class **Type** is the central concept in the FAMIX meta model used to represent types in an object-oriented language. A **Type** can have **Methods** and **Attributes**. A **Type** is defined in a container which can be a **Namespace**, another **Type**, or a **Method**. A **Namespace** contains an arbitray number of **Types**, whereas a Type can only be contained in exactly one **Namespace**. One specialization of Type is **Class**. **Class** represents an entity which can build new instances. Since a **Class** is a specialization of a **Type**, it can also inherit from other classes and can hold attributes and methods. FAMIX does not model interfaces explicitly as meta classes. Instead, a **Class** can represent an interface by setting the value of the <u>**isInterface**</u> to true.

The FAMIX meta model distinguishes between **StructuralEntity** and **BehavioralEntity** (meta classes not shown in the Figure). Instances of **StructuralEntity** denote basic data structures in the source code that have a declared type. Attributes, parameters, and local variables are examples of structural entities. Instances of **BehavioralEntity** define the behavior of the software system, e.g., methods are behavioral entities.

Most object-oriented languages provide mechanism to enrich the source code with meta information, called annotations. Annotations are also an important means to add architecture information to a source code element. That is why, annotations can be used to classify the architecture role of a source code element. The FAMIX meta model defines the class **AnnotationType** to represents the type of an annotation. The concrete instance of an annotation type is represented by the meta class **AnnotationInstance**.

The FAMIX meta model provides the abstract superclass **Association** for relationships between FAMIX entities. For example, **Access** represents an access to a variable by a behavioural entity (for example a method), whereas **Invocation** represents the invocation of a signature on a receiver. An invocation is related with a sender which is a **BehavioralEntity** (e.g. a method) that sends the message. Moreover, it stores the receiver, a structural entity (e.g. a variable) that receives the message. Another attribute of the invocation stores a list of candidates (instances of **BehavioralEntity**) that are potentially invoked.

An instance of a **Type** can have multiple subtypes or supertypes. These are modeled by means of **Inheritance** instances. The inheritance class refers to a pair of types that are related with each other via an inheritance relationship. The inheritance class connects a subtype with its supertype.

7.2.4. FAMIX Ontology

In order to translate the FAMIX meta model to an ontology, it needs to be considered how its constituent parts are described. As mentioned previously, the FAMIX meta model is described using a graphic notation in terms of a UML class diagram to denote classes, their attributes, and associations between them. UML class diagrams and ontologies, share a set of core features. That is why, some of the features of UML class diagram can be directly translated to OWL. As reported by other authors, UML class diagrams can be translated to OWL [BCG05]. Those ideas are adopted to create the FAMIX ontology taking the FAMIX meta model as a reference.

The transformation is manually performed according to a small set of transformation rules. Please note that these rules are not specific to the FAMIX meta model. These rules can be reused for any source code artifact type that is described by a class-diagram-based meta model. Those rules specify which OWL axioms need to be created for each construct considered in the UML class diagram that specifies the FAMIX meta model. For the transformation rules, all relevant constructs are considered with which the FAMIX meta model is defined. Those are the following:

Classes: Classes denote element types classifying objects with the same characteristics and to represent a concept of the domain. For example, **Type** is described as a UML class.

Attributes: Meta model classes define fields to describe properties of a meta class.

Generalization between Classes: As described in the previous section, the meta model defines a class hierarchy that defines generalizations between classes, e.g., a **Class** is a **Type**.

Associations: In the FAMIX Meta Model, associations between classes are solely binary and bidirectional. The FAMIX meta model does not use association classes. Some associations are labeled with a multiplicity and are named by a role[2].

Other constructs of the UML class diagram are not used to specify the FAMIX meta model (such as association classes or generalization between associations). That is why, they are also not considered for the transformation.

[2]Here, the term *role* should not be confused with the meaning of a role in description logics. In this context, a role in a UML class diagram is used to denote how an object participates in a relationship represented by an association [Pen03]

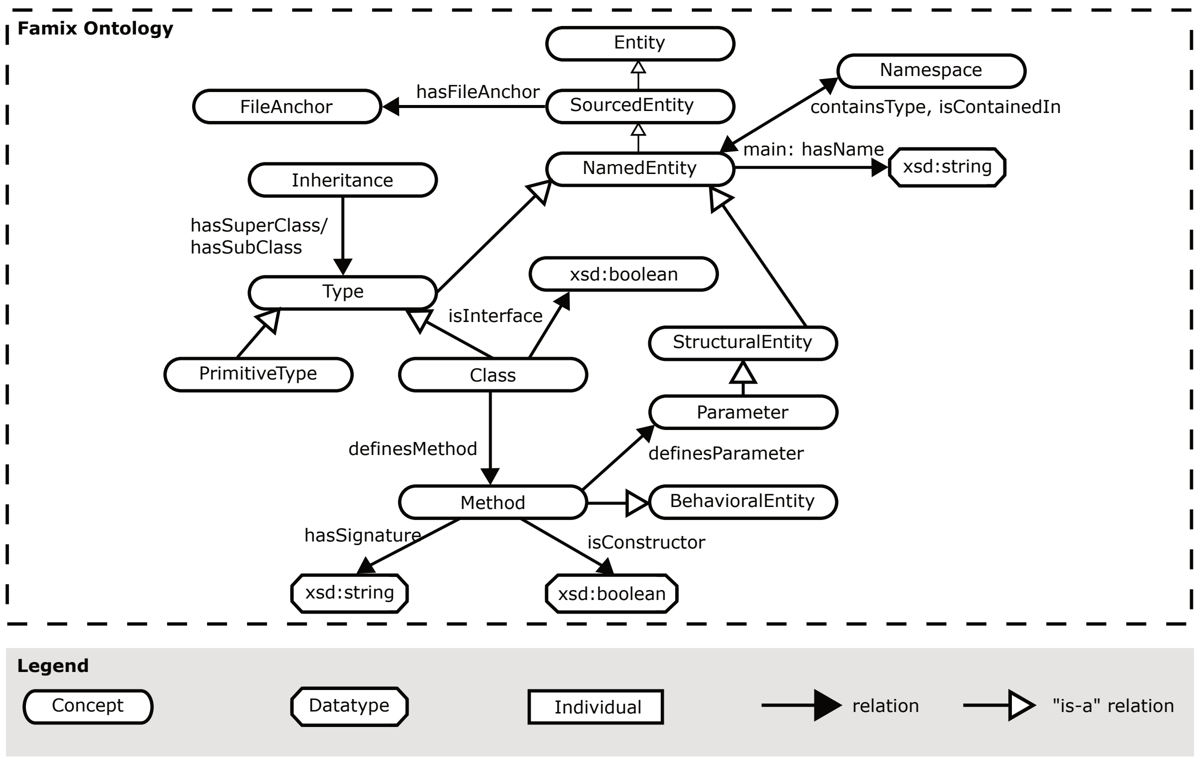


Figure 7.5.: *An excerpt of the FAMIX ontology. The ontology has been transformed according to the transformation rules described in Table 7.2. The entire definition of the ontology can be found in Appendix E.*

Next, the transformation rules are defined and presented. The FAMIX ontology is described with OWL. For each relevant construct of the meta model the transformation to OWL is described. The OWL constructs created in the transformation rules are specified using the functional-style syntax as used in Section 2.4.1. Table 7.2 summarizes all transformation rules that have been applied to transform the FAMIX meta model to the FAMIX ontology. An excerpt of the FAMIX ontology is depicted in Figure 7.5. A concrete transformation based on these is presented in Appendix E. The entire FAMIX ontology is specified in Appendix E.

Mapping Meta Model Classes to OWL classes

Both UML classes and OWL classes, i.e., ontology concepts, denote sets of objects. That is why, the classes of the FAMIX meta model can be mapped directly to atomic concepts [BCG05]. Therefore, for each class in the FAMIX meta model a class declaration in OWL is created in order to create the corresponding ontology concept. The *name* of a class provides the name of the OWL class:

$$Declaration(Class(name))$$

In the FAMIX meta model, some abstract classes – such as **Type** – are defined. Abstract classes require that no instances can be created of this classes. Those classes cannot be translated to OWL, since OWL does not support such a construct. That is why, abstract classes are simply translated to OWL classes and the abstract property of the class is omitted in the translation.

Table 7.2.: *Summary of the transformation rules (TR) applied to transform the FAMIX meta model to its ontology.*

Rule ID	Description	OWL Axiom in Functional-style Syntax
	Meta Model Class	
TR 1	Specify a class declaration axiom for class *name*.	*Declaration(Class (name))*
	Attributes	
TR 2	Specify a declaration axiom for a datatype property A for the attribute.	*Declaration(DatatypeProperty(A))*
TR 3	Specify the domain C of the datatype property A.	*DataPropertyDomain(A C)*
TR 4	Specify the data type range *type* of the datatype property A.	*DataPropertyRange(DatatypeProperty(P type)*
	Generalization	
TR 5	Specify a *SubClassOf* axiom for the generalization between meta model classes C and D.	*SubClassOf(C D)*
TR 6	Specify sibling concepts to be pairwise disjoint.	*DisjointClasses(C_i C_j)*, $i = 1,...,n;\ j = 1,...,n;\ i \neq j$
	Associations	
TR 7	Specify a declaration axiom for object property R.	*Declaration (ObjectProperty(R))*
TR 8	Specify a declaration axiom for object property S and define it to be inverse with R.	*Declaration (ObjectProperty(S))* *InverseObjectProperties (R S)*
TR 9	Specify the domain C and range D for properties R and S.	*ObjectPropertyDomain(R C)*, *ObjectPropertyRange(R D)* *ObjectPropertyDomain(S D)* *ObjectPropertyRange(S C)*
TR 10	Optional: In case the multiplicity equals 1, specify R to be a functional property.	*FunctionalProperty(R)*

Mapping Attributes to Datatype Properties

Attributes of a meta class store properties of a class. Such an attribute is defined by a name and a data type. Attributes of a meta class are mapped to OWL datatype properties. This means, for each attribute A of a meta class the following declaration is specified:

$$\textit{Declaration(DatatypeProperty(A))}$$

Similar to object properties (representing associations), the domain and range constraints need to be set. The domain refers to the meta class A for which the attribute is defined:

$$\textit{DataPropertyDomain(A C)}$$

The range of the data type property refers to the data type defined for the attribute. Data types are transformed to the corresponding *XSD* [xsd] data type definition provided by OWL. The FAMIX meta model uses the String, Boolean, and Number data types for attributes. The String and Boolean data types can be directly mapped to their XSD representatives (`xsd:string` and `xsd:boolean`). The Number data type represents a data type that stores unsigned integer values, e.g., for line numbers. This data type is mapped to `xsd:unsignedLong`. The data type needs to be specified in the range axiom:

$$\textit{DataPropertyRange(A }\texttt{xsd:string}\textit{)}$$
$$\textit{DataPropertyRange(A }\texttt{xsd:boolean}\textit{)}$$
$$\textit{DataPropertyRange(A }\texttt{xsd:unsignedLong}\textit{)}$$

Attributes are functional, meaning that an individual can only have at most one value for this attribute. For example, a class can only have one name specified. Therefore, a *FunctionalProperty* axiom must be created for each data type property representing a class attribute:

$$\textit{FunctionalProperty(A)}$$

Some attributes have the "*" multiplicity specified, i. e, no restriction on the number of elements is defined. For those, no additional axioms need to be defined.

Attributes are unique within a class, but two classes may have two attributes with the same name and also possibly have different types. This needs to be considered during the translation, since properties (and classes) are globally defined and uniquely identified by URI references. It needs to be checked whether there already exists such a property in the ontology. The name of the data type property must be chosen appropriately. Alternatively, the domain of the property must consider the union of the concepts that define attributes with the same name.

Mapping Generalization to Subclass Axioms

The FAMIX meta model describes class hierarchies between meta model classes to denote "is-a" relationships. For example, a **Class** is a **Type**. Generalization is supported by the subsumption relation in description logics and OWL (see Section 2.4.1). The inheritance between OWL classes corresponds to inheritance between UML classes due to the semantics of $\sqsubseteq$. That is why, generalization between two classes C and D in the FAMIX meta model is translated to a subsumption axiom, i.e., $C \sqsubseteq D$, where C and D are atomic concepts corresponding to the classes C and D:

$$\textit{SubClassOf(C D)}$$

Although not explicitly enforced in the FAMIX meta model, all sibling concepts C_i inheriting from the same concept are defined to be pairwise disjoint with each other in order to avoid undesired inferences:

$$\mathit{DisjointClasses}(C_i\ C_j),\ i = 1,...,n;\ j = 1,...,n;\ i \neq j$$

For example, it is necessary to state that a *Class* – the concept representing a **Class** – and a *Method* – the concept representing a **Method** – are disjoint to each other. Both are sub-concepts of the concept *ContainerEntity*. An individual can never be a *Class* and a *Method* at the same time, i.e., a code element is either a class or a method.

Mapping Associations to Object Properties

Each association R in the FAMIX meta model between a meta model class C and another meta model class D is represented by an object property in the FAMIX ontology. That is why, a declaration for object properties needs to be specified:

$$\mathit{Declaration(ObjectProperty(R))}$$

Since the associations in the meta model are bidirectional, a corresponding inverse role S is defined. This means, an additional declaration for object properties needs to be specified for this role:

$$\mathit{Declaration(ObjectProperty(S))}$$

In order to denote that this property is inverse to R, an *InverseObjectProperties* axiom needs to be declared for the properties:

$$\mathit{InverseObjectProperties(R\ S)}$$

All associations are denoted by explicit role names in the FAMIX meta model. However, not both association ends are named. For the other association role name, a name is chosen so that it appropriately represents the inverse property. For example, for the **contains** association - which is an association between a namespace and a type the namespace contains - its corresponding containsType property and the respective inverse property isContainedIn are defined.

In a next step, the scope of the object property R in terms of domain and range constraints is specified. This means, the following declarations are created for the OWL classes C and D corresponding to meta classes that are connected by an association in the FAMIX meta model:

$$\mathit{ObjectPropertyDomain(R\ C)}$$
$$\mathit{ObjectPropertyRange(R\ D)}$$

The domain range constraints are specified accordingly for the inverse property S.
Associations can be annotated with multiplicities. The FAMIX meta model defines the multiplicities 1 and * for associations. In the FAMIX meta model, when no multiplicity is explicitly specified, it equals 1. Other multiplicities are not defined in the meta model. For the multiplicity *, no further cardinality axioms need to be defined, since there is no restriction for the number of the relations that must exist between two individuals. In case the multiplicity of an association end equals 1, a *FunctionalObjectProperty* axiom is specified for the corresponding object property:

$$\mathit{FunctionalProperty(R)}$$

7.2.5. Source Code Facts Based on the FAMIX Ontology

Source code is represented as individuals of the FAMIX source code ontology. This means, for example, object-oriented classes defined in the source code are represented as individuals of the CLASS concept or a method defined in a class is defined as an individual of the concept METHOD.

In order to illustrate how source code is represented with the FAMIX ontology, an exemplary code snippet is used. The example is shown in Figure 7.6. Figure 7.6 a) shows a code snippet that implements a service-oriented interface [Erl05]. Figure 7.6 b) depicts an excerpt from the FAMIX meta model including the meta classes used to represent the code snippet in Figure 7.6 a). In Figure 7.6 c) the corresponding individuals describing the code using concepts from the FAMIX-based code ontology are shown. In order to express the code, ontology concepts CLASS, METHOD, INHERITANCE, NAMESPACE, PRIMITIVETYPE, and PARAMETER are needed. The statements in Figure 7.6 c) express that the method `getStoreStockReport` is contained in the interface `IReporting` using the role parentType. The ontology additionally defines the object property definesMethod in order to represent the relation between an individual of a CLASS concept and an individual of the METHOD concept. In the code snippet, the interface `IReporting` extends the `Remote` class. This is modeled by an individual of the INHERITANCE concept. Since interfaces are not modeled as an explicit element, but as a property of the CLASS element in the FAMIX meta model, the isInterface data type property is used and set to “true” to denote that a class is an interface. Figure 7.6 d) represents the corresponding graph representation of the ontology-based source code representation.

It is important to note that the relation hasName is reused by the FAMIX ontology. This relation is originally defined in the source code artifact ontology (prefix “main”) and imported by the FAMIX ontology. Since a SOURCEDENTITY, e.g., a NAMESPACE or a CLASS, is a SOURCECODEARTIFACT that has a name, this relation also applies to individuals of the SOURCEDENTITY concept.

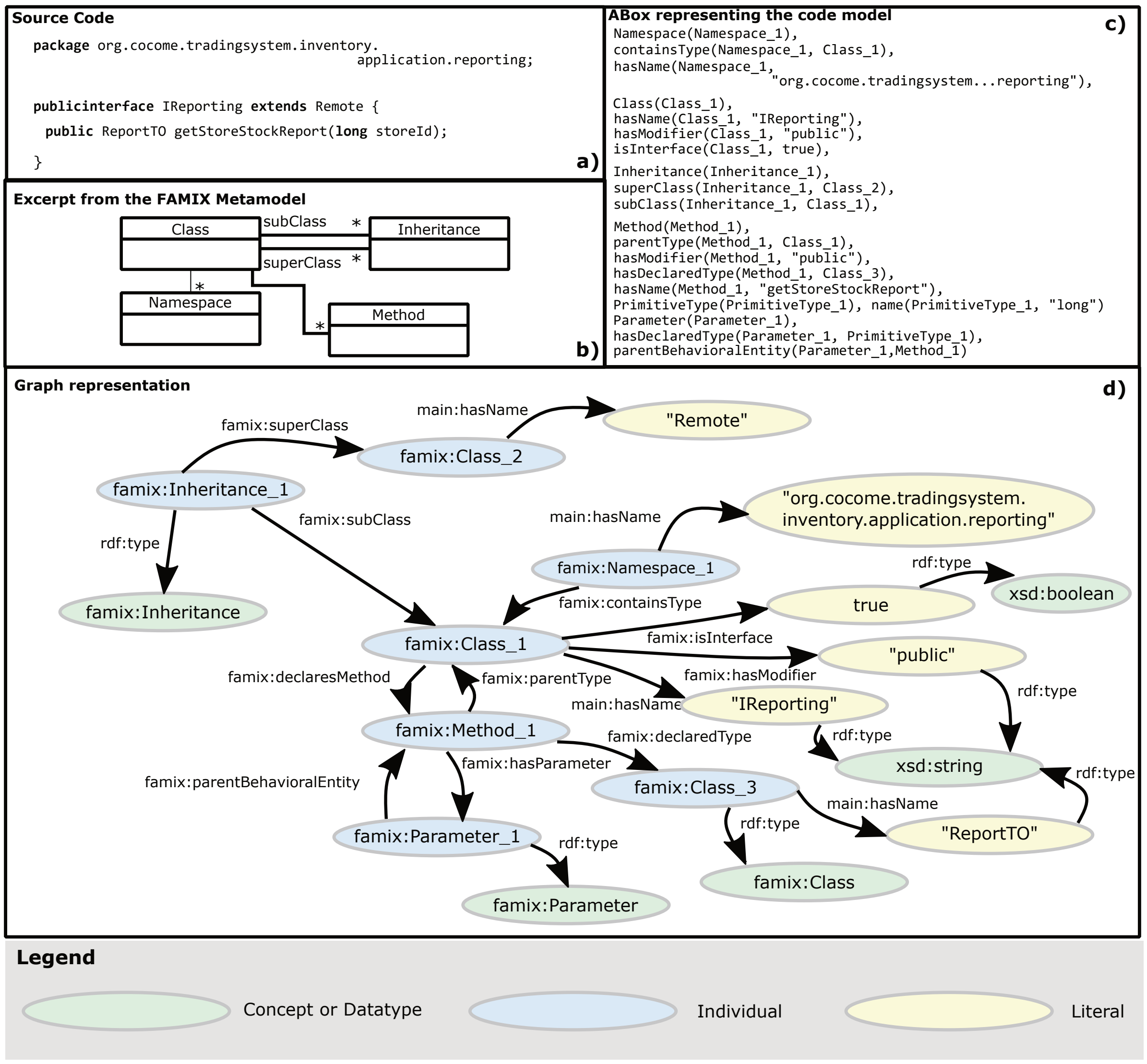
7.2.6. Maven Ontology

Maven [mav] is a software project management tool. With Maven, developers can specify and manage the build lifecycle of their software project. The ontology is shown in Figure 7.7.

A central concept in the ontology is the MAVENPROJECT. A MAVENPROJECT is a Java project that contains a so-called Project Object Model (POM) - represented by the concept MAVENPOMFILE - an XML file with a predefined structure with which the build process is specified (containsPOMFile). As a MAVENPOMFILE is defined to be a sub-concept of the FILE concept of the general source code artifact ontology, it is associated with a file path (hasPath). Here, the hasPath relation is imported and reused from the general source code ontology (Section 7.2.1).

MAVENPROJECTS are built as so-called MAVENARTIFACTS. A MAVENARTIFACT is defined by so-called Maven coordinates that encompass a group ID (hasGroupId), an artifact id (hasArtifactId) and a version (hasVersion). Together, they uniquely identify the MAVENARTIFACT. The type of the artifact that should be build, i.e., its packaging (hasPackaging), is defined in the pom file (MAVENPOMFILE producesArtifact MAVENARTIFACT). For example, a POM file can define that a project is built as a Java archive [jar] (Jar) file.

In Maven, the packaging types are predefined, e. g., *pom*, *jar*, or *maven-plugin*. That is why, a


Source Code

```
package org.cocome.tradingsystem.inventory.
                                 application.reporting;

publicinterface IReporting extends Remote {
  public ReportTO getStoreStockReport(long storeId);

}
```

ABox representing the code model

```
Namespace(Namespace_1),
containsType(Namespace_1, Class_1),
hasName(Namespace_1,
                  "org.cocome.tradingsystem...reporting"),

Class(Class_1),
hasName(Class_1, "IReporting"),
hasModifier(Class_1, "public"),
isInterface(Class_1, true),

Inheritance(Inheritance_1),
superClass(Inheritance_1, Class_2),
subClass(Inheritance_1, Class_1),

Method(Method_1),
parentType(Method_1, Class_1),
hasModifier(Method_1, "public"),
hasDeclaredType(Method_1, Class_3),
hasName(Method_1, "getStoreStockReport"),
PrimitiveType(PrimitiveType_1), name(PrimitiveType_1, "long")
Parameter(Parameter_1),
hasDeclaredType(Parameter_1, PrimitiveType_1),
parentBehavioralEntity(Parameter_1,Method_1)
```

Figure 7.6.: *Source code represented with the FAMIX-based source code ontology: a) The Java source code snippet. b) An excerpt of the FAMIX meta model with the meta classes and associations used in this example. c) An excerpt of the ABox representing the code snippet based on the concepts and relations of the FAMIX ontology. d) The graph representation of the ABox. Nodes represent concepts, datatypes, individuals, or literals. Connection between nodes represent relations. The prefix* `famix` *represents the URI of the FAMIX code ontology (see Table 7.1). The prefix* `main` *refers to the namespace of the source code ontology. Please note that this graph only represents an excerpt of the ontology-based representation of the code snippet. Not all concepts, relations, and individuals are shown here.*

concept MavenPackagingType with corresponding individuals is defined. Instead of simply storing the packaging type as a string value, it is stored as an individual of this concept. This means, during fact extraction, no new individuals of MavenPackagingType are created, but the predefined individuals are re-used. Additionally, it prevents that invalid values for the packaging are used.

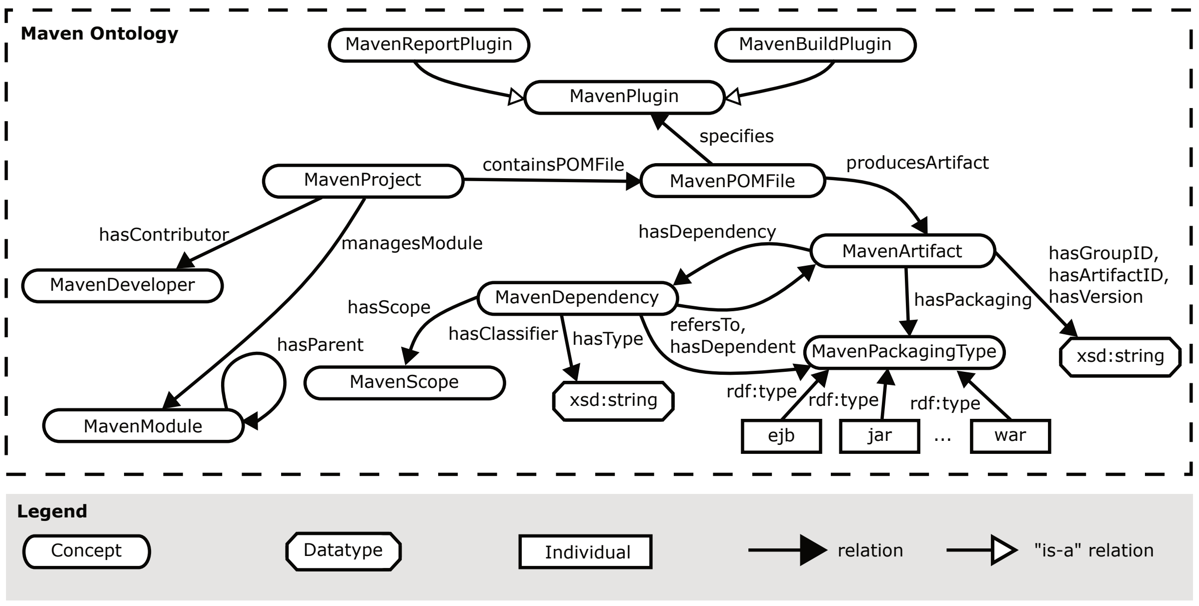


Figure 7.7.: *Excerpt of the Maven Ontology and its main concepts and relations..*

A MavenProject has contributors (hasContributor) who are MavenDevelopers.

A MavenProject is organized as a MavenModule, this means a MavenProject manages a MavenModule (managesModule). A MavenModule is a sub-project of a MavenProject. That is why, Maven projects are in a hierarchical structure. This means, MavenArtifacts may have a parent MavenModule (hasParent).

A MavenArtifact has dependencies to other artifacts that refer to other projects or libraries that are necessary in order to implement the functionality (hasDependency). Maven dependencies are modeled as individuals of the concept MavenDependency. Modeling dependencies as instances of a class explicitly - and not as object property - allows to define additional attributes for this class: A MavenDependency refers to an artifact (refersToArtifact) that is described by a group id, artifact id, and a version. A MavenDependency has a dependent MavenArtifact that uses this dependency (hasDependent). Moreover, a MavenDependency defines a classifier (hasClassifier) and a scope (hasScope). The classifier allows to distinguish artifacts that are built from the same POM, but differ in their content, e.g., an artifact that contains the source code and an artifact that only contains the Javadoc [jav].

The scope attribute in a pom determines how transitive dependencies should be handled. For the scope, a concept MavenScope is defined. The concept is associated with predefined individuals that encode the possible values for the scope, namely compile, provided, runtime, test, system, and import. A MavenDependency additionally specifies the type of the artifact, i.e., the packaging, that is used as dependency (e.g. jar, war, pom etc.) (hasType). The individuals MavenPackagingType as described above can be used here.

The functionality of Maven is configured by plugins (MavenPlugin) that are responsible for executing the actual build tasks. Those plugins need to be specified in the MavenPOMFile (usesPlugin). Maven defines BuildPlugins and ReportingPlugins that are configured differently. A BuildPlugin has an artifact id, a version, and defines several executions, while a ReportingPlugin defines several ReportSets.

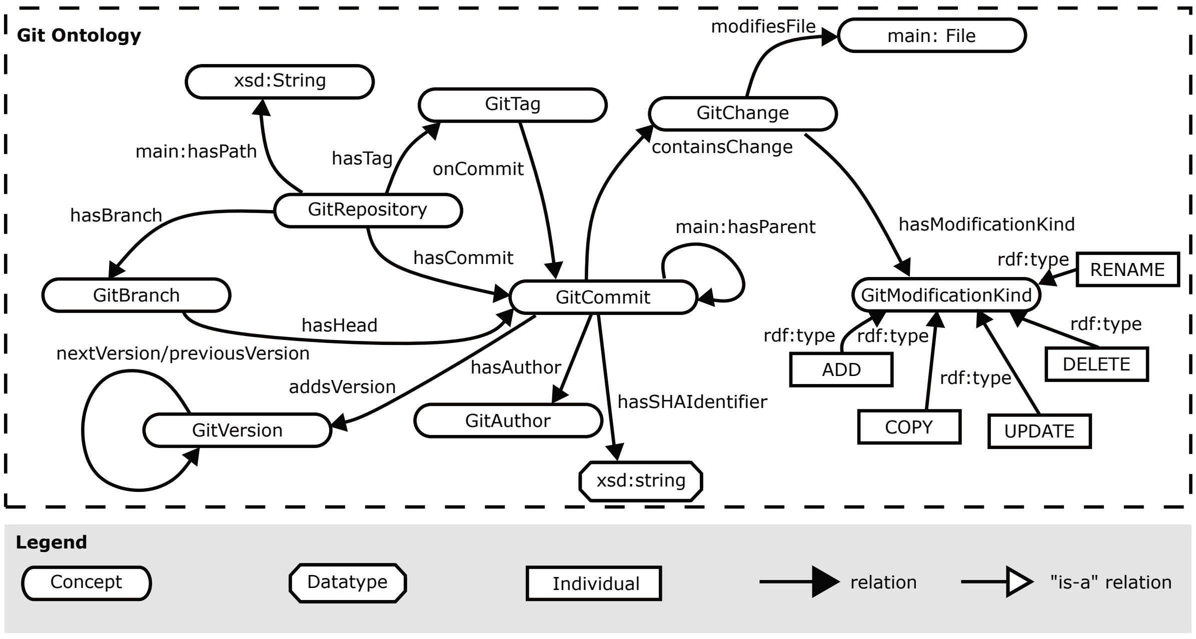


Figure 7.8.: *An excerpt of the Git ontology and its main concepts and relations.*

7.2.7. Git Ontology

The Git ontology allows for the modeling and preserving of historical information from version control systems based on Git [git]. Moreover, it reuses concepts and relations from the general source code artifacts ontology. Figure 7.8 depicts an excerpt of the Git ontology. In this approach, the Git ontology is used to store historical information of architecture violations.

A GitRepository contains several GitCommits. The concept GitCommit represents a commit that summarizes and saves current changes made to one or more files in a GitRepository where each is uniquely identified by an SHA identifier (hasSHAIdentifier). A GitCommit contains one or more changes (containsChange GitChange). In each change, one or more Files are modified by an author (GitAuthor). A File is, for example, a Java class, a Maven POM, or a text file. The author provides a message for each GitCommit he performs (hasMessage). An author has a name and an e-mail address specified. Every GitCommit has a relation to at least one parent commit (hasParent), except the initial commit which has no parent commit. Note that the relation hasParent is re-used from the main ontology which is applicable on all types of Artifacts. Since a GitCommit is defined to be a sub-concept of Artifact, hasParent is applicable on this concept. A GitChange has a specific modification kind (GitModificationKind). The ontology defines the modification kind Add, Copy, Delete, Rename, and Update as individuals GitModificationKind.

A GitBranch points to an independent line of development. A GitBranch has a head which points to the latest commit of this branch (hasHead).

A GitTag is a designated label of a commit. Such a tag is normally used in order to denote a release version in the repository. A GitTag is associated with a GitCommit using the onCommit property.

When performing a GitCommit, a new version is added to the repository. A version is represented by the concept GitVersion.

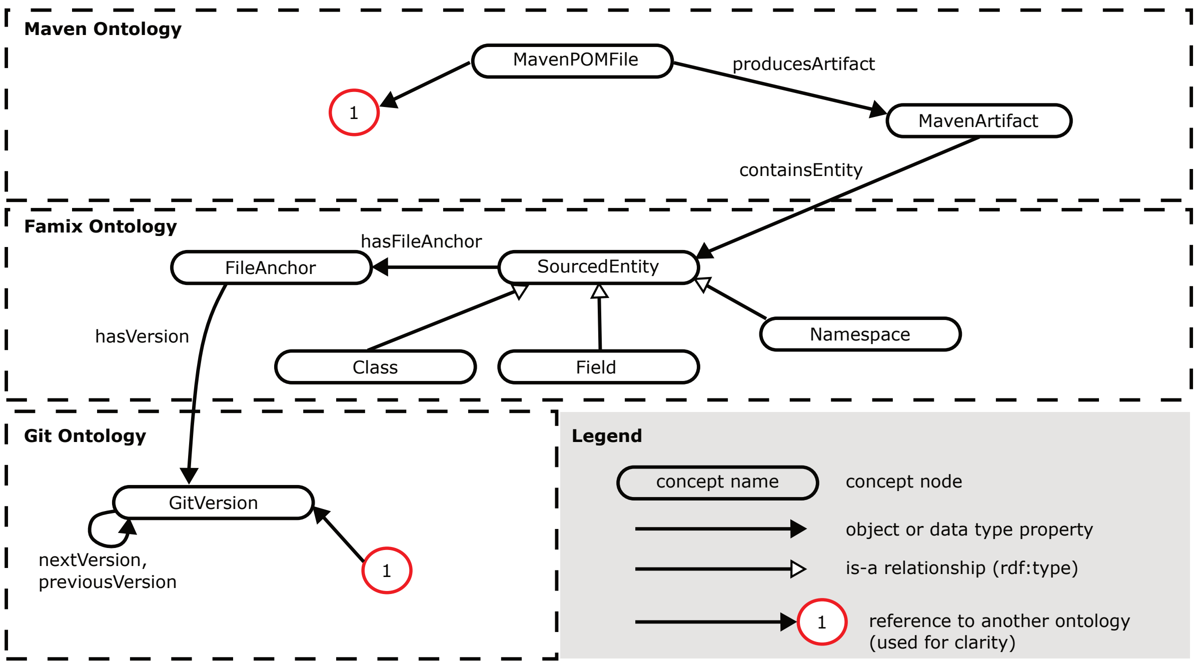


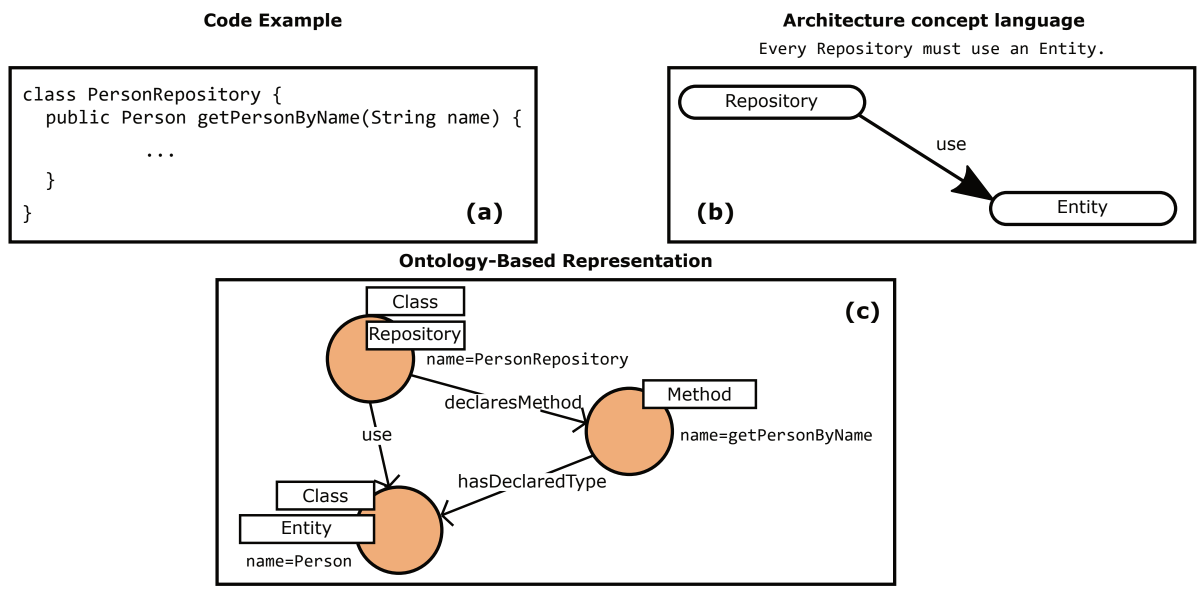
Figure 7.9.: *Integrating the ontologies by specifying explicit relationships between their concepts.*

7.2.8. Integrating the Ontologies

Since different types of source code artifacts are using the same underlying representation, namely ontologies, they can be integrated with each other. This is done by defining explicit relationships between concepts defined in different ontologies. For example, the Git and FAMIX ontology can be connected with each other. This is useful in case a class should be enriched with historical information. Another use case is to connect a Maven artifact with the corresponding Java Package, i.e., in a NAMESPACE, by which it is implemented. Figure 7.9 visualizes exemplary relationships that are defined between the ontologies. A MAVENARTIFACT from the Maven ontology is connected with the concept SOURCEDENTITY. This means that a MAVENARTIFACT can contain anything that inherits from this concept, e.g., a class, a namespace, or a field. A FILEANCHOR in the FAMIX ontology connects a SOURCEDENTITY with a concrete file. This file can have a specific version if it is maintained in a (Git) version control system. The same can be defined for a MAVENPOMFILE.

7.3. Rule-based Detection of Architecture Concepts and Relations in Code

Architecture concepts and relations used to describe architecture are not visible on an implementation level. This means that architecture rules cannot be directly validated on the source code, since it uses low-level abstractions. For example, on an architectural level, the software architect describes the system using concepts like *components* or *ports*, while programming languages only provide low-level concepts like *class* or *method*. Defining an architecture-to-code-mapping is therefore necessary in order to identify architecture concepts and relations in the source code and to extract the implemented architecture (see Section 2.1). As revealed in the empirical



```
class PersonRepository {
  public Person getPersonByName(String name) {
          ...
  }
}
```

Figure 7.10.: *An ontology-based representation of a code snippet as a graph. Nodes represent concept assertions and edges represent role assertions. Rectangles associated with a node are used to represent the corresponding concept an individual belongs to. Labels with* `name=` *designate a property of the individual. In this case, each node is associated with a name (meaning a method name or class name).*

study in Chapter 3, such a mapping is necessary to make architecture visible in the code which is an important goal of architecture enforcement. By applying the mapping, the source code model is enriched with additional information about architecture concepts and relations, and consequently, the abstraction level of the source code is raised onto the architecture level. In a subsequent step, architecture rules defined for architecture concepts can be validated on the extracted implemented architecture.

The core idea here is to represent architecture concepts in code structures by applying this mapping on the source code, or, more precisely, on its ontology-based representation. Figure 7.10 illustrates this idea using an exemplary code snippet. The code snippet shows a class that implements the architecture concept REPOSITORY. The corresponding ontology-based representation of this code snippet is shown as a graph, where nodes represent the concept assertions and edges represent role assertions. Each node is labeled with the name of the concept it belongs to. The architectural role of this class (i. e., the architecture concept it implements) can be identified based on the suffix of its name. If its name ends with the word “Repository”, the corresponding architecture concept is assigned to this class. As can be seen, the node representing the Java class `PersonRepository` is additionally labeled with the concept *Repository*. This means that it is not only an individual of the low-level code concept *Class*, but it is also an individual of the architecture-level concept *Repository*. The code-level relation between the individuals is also lifted to the architectural level. In this example, the class `PersonRepository` defines a method that returns a `Person` object. This means that the `PersonRepository` class is connected to the Person class via the architecture relation *use*.

7.3.1. Modeling Mapping Rules in SWRL

Mapping rules are formalized as SWRL rules. As described in Section 2.4.7, an SWRL rule consists of a head and a body. In the context of architecture-to-code-mapping, the head contains the architecture concept or relation that should be derived from the code model. The body specifies a set of conditions combined by conjunctions that must be satisfied so that the architecture concept or relation can be derived from the code model. For each architecture concept and relation at least one mapping rule needs to be defined. Otherwise, the architecture rules defined for architecture concepts cannot be validated on the code, because the architectural information is missing. Additionally, only architecture concepts and relations defined in the architecture ontology can be used in the mapping rules. Undefined concepts and relations cannot be inferred. The formalization of mapping rules comprises two parts, namely the *architecture concept mapping* and the *architecture relation mapping*. In the following, the parts are described in more details.

Architecture concept mapping

An architecture concept mapping infers for an individual from the code model whether it is an instance of an architecture concept defined in the architecture ontology. In order to define the mapping rule, the following steps need to be performed:

1. **Select** the architecture concept A that should be inferred from the code and formalize it as a unary atom $A(x)$ in the head of the rule, where A is the architecture concept to be inferred and x is a variable representing an individual of the code model.

2. **Decide** for the properties that need to be fulfilled by x and other properties that need to be satisfied in the code model in order to infer the concept for the code element. Properties are concepts or relations from a code ontology or from the architecture ontology. Only known code concepts and relations, i.e., concepts and relations that have been imported to the mapping ontology, can be used. These are concepts and relations defined in the artifacts ontologies presented in the previous sections. Additionally, concepts and relations from the architecture ontology can be used as properties. When undefined concepts and relations are used, the architecture concept cannot be inferred for x.

3. **Formalize** the properties as atoms.

4. **Combine** the properties - represented as atoms - selected in the previous steps by conjunction. They represent the body, i.e., the antecedent, of the rule.

Example 7.3.1: Mapping Architecture Concepts

In the following, an exemplary mapping rule for the Repository concept from Figure 7.10 is defined using SWRL. Informally, the rule can be stated as follows:

If something is a CLASS (property) whose name ends with *"Repository"* (property), then it is classified as a REPOSITORY (architecture concept).

The code concept CLASS and the code-level relation hasName which encodes the name of the class are properties that need to be satisfied by an individual. They are defined in the FAMIX ontology. The rule can then be formalized in SWRL as follows

$$Class(x), hasName(x, name), endsWith(name, ''Repository'') \rightarrow Repository(x)$$

The variable x corresponds to the individual to which the architecture concept CLASS is assigned to. The atoms $Class(x)$, $hasName(x, name)$, and $endsWith(name, ''Repository'')$ correspond to the properties the individual must satisfy. $endsWith$ is a built-in function of SWRL. If an individual is found satisfying these properties, it is bound to x and consequently assigned to the architecture concept REPOSITORY. A reasoner which is applied on this rule will assign the concept *Repository* to the individual represented by the individual x and adds the assertion to the knowledge base.

Architecture relation mapping

An architecture relation mapping infers architecture relations between two individuals from the code model. For each architecture relation defined in the architecture ontology, at least one mapping rule has to be defined. The steps performed for architecture relation mapping are as follows:

1. **Select** the architecture relation r that should be inferred from the code and formalize it as a binary atom $r(x, y)$, where x and y are variables representing individuals from the code model between which the architecture relation should be established. The atom is added to the head of the rule. It will be added as a role assertion to the code model if the conditions in the body are satisfied.

2. **Decide for** properties that need to be fulfilled by x and y and by other individuals of the code model in order to infer the architecture relation between x and y. According to the concept mapping, only defined concepts and relations from the code and the architecture ontology can be used. Otherwise, the architecture relation cannot be inferred for x and y.

3. **Formalize** the properties as atoms.

4. **Combine** the atoms representing the properties by conjunction and add them to the body of the rule.

Example 7.3.2: Mapping Architecture Relations

Applying the steps as described previously, the exemplary architecture relation from Figure 7.10 can be derived from the code as follows:

$$declaresMethod(x,m), Class(x), hasDeclaredType(m,y), Class(y) \rightarrow use(x,y)$$

In this mapping rule, the *use* relation between two individuals representing classes (concept CLASS from the FAMIX ontology) is derived. A *use* relation between two individuals x and y is derived when x is a CLASS that declares a method m that has the declared type y which is also a CLASS.

In the next section, it is shown how mapping rules can be classified according to so-called mapping conventions. Each mapping convention is defined by the kind of information it uses to extract the implemented architecture from the source code. In a next step, those mapping conventions are formalized using SWRL that allows to apply the mapping on the ontology-based representation of the source code in order to extract the implemented architecture.

7.3.2. Classifying Mapping Conventions to Identify Architecture Concepts

Software architects and developers provide hints about the software architecture – consciously or unconsciously – in the code in order to express the design intent. For example, those hints capture information about the implemented architectural style, constraints, components, and their properties. By doing so, they adopt a so-called *architecturally-evident coding-style* [FG10]. This information encoded with such a coding style can be exploited in order to identify architecture concepts and relations in the source code.

Such an architecturally-evident coding style may use the following patterns to embed the design intent in the source code. The patterns use concepts and relations from the source code artifact ontologies defined in the previous sections.

Name Conventions Assuming that architects and developers have consciously or unconsciously used naming conventions to denote the architectural role of a source code element (e.g., for classes), the names can help to extract architecture concepts from the code. In the previous example, it was assumed that classes implementing the repository pattern [Fow02] have a name that ends with "Repository". This is expressed by the following SWRL rule:

$$Class(c),\ name(c,name),\ swrl:endsWith(name,''Repository'') \rightarrow Repository(c)$$

Metadata Some programming languages provide mechanisms to enrich the source code with metadata information. For example, Java provides so-called annotations. A lot of frameworks like Spring [spr] and JEE [jee] use annotations to define the architectural role of a class, e.g., Controller, Service, Repository, EJB, MessageDriven, Entity etc. Using metadata information given by Java annotations, the following SWRL rule assigns the *Repository* role to a class:

$$\begin{gathered}Class(c), hasAnnotationInstance(c,a), AnnotationInstance(a),\\ hasAnnotationType(a,at), name(at,''Repository'') \rightarrow Repository(c)\end{gathered}$$

Package Structure The package structure of the implementation may indicate how the software system is structured into components. One package may implement a component or a similar architectural abstraction. This can be formalized by the following SWRL rule:

$$Namespace(n) \rightarrow Component(n)$$

Maven projects can also be considered as components:

$$MavenProject(n) \rightarrow Component(n)$$

Class Hierarchy The architectural role of a class may be indicated by the interfaces it implements or the base classes it extends as by the following rule:

$$\begin{aligned}&Class(c), Class(c2), Inheritance(i), hasSubClass(i, c),\\&hasSuperClass(i, c2), name(c2,'' Repository'') \rightarrow Repository(c)\end{aligned}$$

Package Containment The architectural role of a class can be identified by its package location. For example, a class must be located in a specific package, e.g., classes that are located in a package `*.domain.*` are classified as an *Entity*:

$$\begin{aligned}&Class(c), Namespace(p), isContainedInNamespace(c, p),\\&name(p, name), swrl : contains(name,'' domain'') \rightarrow Entity(c)\end{aligned}$$

7.4. Computing Architecture Violations

In order to compute architecture violations, two major steps are performed. First, the knowledge base is transformed so that it can be evaluated under the CWA and second, the output of the reasoner is processed. In the following, these two steps are explained in more detail:

1) Enforcing the Closed World Assumption: By default, description logics and OWL adopt the OWA. It assumes that information is incomplete and therefore allows for validating models that miss some information. However, in the context of architecture conformance checking, a CWA is more appropriate. In Section 2.4.6 the distinction between OWA and CWA have been already described. CWA is enforced by closing the domain, i.e., stating that the set of all individuals in the domain coincide with the set of individuals explicitly mentioned in the ontology. This means that only the individuals that represent the source code and the implemented architecture are considered, respectively. For this, the top OWL class THING is defined to be equivalent to the set of individuals defined in the knowledge base. OWL provides the *Enumeration* construct for this. Additionally, the UNA is applied. It requires that individuals that have different names are also considered as different individuals. OWL provides the *DifferentIndividuals* construct in order to explicitly declare a set of individuals as different. Having prepared the knowledge base to be evaluated under CWA, the reasoner can be applied on the knowledge base and the output of the reasoner is processed further.

2) Computing Explanations of Inconsistencies: As described in Section 7.1, architecture violations are detected by verifying whether the implemented architecture is consistent with the architecture ontology. An inconsistency in the implemented architecture corresponds to an architecture violation. Inconsistencies are further described by so-called *explanations* that are calculated by the reasoner. Explanations are *minimal entailing subsets of an ontology* [HPS10]. Explanations given by the reasoners provide useful information about which axioms are violated and which assertions are responsible for violations. Explanations calculated by the reasoner are structured as so-called *proof trees* [Krö10b] [EJ85]. An architecture rule has been violated if there exists a proof tree for the corresponding class axiom:

Definition 7.4.1: Violation of an Architecture Rule and the Architecture Concept Language

Let R be the set of architecture rules defined in an architecture concept language. Let PT_r be the set of proof trees that have been calculated by a reasoner for a rule $r \in R$. Further, the function

$$violated : R \rightarrow \{true, false\}$$

indicates for a rule $r \in R$ whether it has been violated ($true$) or not ($false$). Then:

$$violated(r) = \begin{cases} true, & \text{if } PT_r \neq \emptyset \\ false, & \text{if } PT_r = \emptyset \end{cases}$$

An architecture concept language is violated if there exists a rule $r \in R$ that has been violated.

Each proof tree is represented as a structured text according to an hierarchical structure. When referring to a proof tree in the following explanations, its textual representation is meant. The nodes of the tree are classified and labeled according to the three types `VIOLATED`, `ASSERTED`, and `NOT_INFERRED`:

- A node labeled as `VIOLATED` denotes the root of the tree. It stands for the axiom, i.e., the architecture rule, that has been violated.
- A node labeled as `ASSERTED` denotes a concept or role assertion from the knowledge base that causes the axiom to be violated.
- A node labeled as `NOT_INFERRED` refers to missing information in the knowledge base that causes the axiom to be violated.

Each proof tree that has been calculated for an architecture rule is considered as an architecture violation of this rule. A proof tree contains exactly one `VIOLATED` node and at least one `ASSERTED` node. Depending on the rule type that has been violated, it contains at least one `NOT_INFERRED` node. In a proof tree, axioms and asserted or missing facts are written as *subject-predicate-object* triples using a syntax similar to the Terse RDF Triple Language (Turtle) [tur08]. The information contained in a proof tree is used to define and describe an architecture violation of a rule:

Definition 7.4.2: Architecture Violation

An architecture violation of a rule r – defined in the architecture concept language $ACL = (AC, AR, R)$ (Definition 5.1.4) – is a 4-tuple $v = (s_v, p_v, o_v, type)$ where

- $s_v : PT_r \to IA$ is a function that returns the subject of the violation for a proof tree of r, i.e., the code element that causes the violation,
- $p_v : PT_r \to AR$ is a function that returns the predicate for a proof tree of r, i.e., the architecture relation that is established or not established by the subject s_v,
- $o_v : PT_r \to IA \uplus AC$: is a function that returns the object for a proof tree of r, i.e., returns either a) a code element $c \in IA$ with which the subject illegally establishes a relation (divergence) or b) the architecture concept $c \in AC$ the subject misses to establish a relation with (absence).
- $type$ is the type of the violation, i.e., $type \in \{absence, divergence\}$

The following listings present some exemplary proof trees. In order to distinguish the different elements in the following explanations, individuals are emphasized as UNDERLINED CAPITALS, concepts are emphasized as *italics*, and code elements are emphasized in `typewriter` font.

Listing 7.1 shows an exemplary proof tree of the *Can-Only* rule type. In line 1 the corresponding triple presentation of the validated rule is shown. The prefixes `arch` and `code` denote the namespaces of the architecture and code ontologies, respectively. The prefixes are used as abbreviations for the namespace URIs of the ontologies. The reasoner found that the rule "`Every` *Repository* `can only` *use* `an` *Entity*." has been violated. The violation can be explained as follows based on the axioms stated in the proof tree: There is an individual PERSONREPOSITORY in the code model that is classified as a *Repository* (line 2) and that is connected with an another individual ACCOUNTREPOSITORY (line 3) that is, however, not an individual of the concept *Entity* (line 4). Since an individual of the concept *Repository* is only allowed to be related with an individual of the concept *Entity* via the architecture relation *use*, this constitutes a violation. According to Definition 7.4.2, the violation of this rule can be described as the 4-tuple $v = (PersonRepository, use, AccountRepository, divergence)$.

Listing 7.1: *Exemplary proof tree of the Can-Only rule type.*

```
1 VIOLATED arch:Repository rdfs:subClassOf (arch:use only arch:Entity)
2 ASSERTED code:PersonRepository a arch:Repository
3 ASSERTED code:PersonRepository arch:use code:AccountRepository
4 NOT_INFERRED code:AccountRepository a arch:Entity
```

In contrast, Listing 7.2 shows an example that does not contain a `NOT_INFERRED` node. The exemplary rule "`No` *Entity* `can` *use* `a Repository`." is evaluated (negation rule type). Line 1 shows the corresponding triple representation of the rule. It is found that an individual that is classified as an *Entity* uses an individual that is classified as a *Repository* which is forbidden according to the rule. According to Definition 7.4.2, the violation of this rule can be described as the 4-tuple $v = (PersonEntiy, use, PersonRepository, divergence)$.

Listing 7.2: *Exemplary proof tree of the negation rule type.*

```
VIOLATED arch:Entity rdfs:subClassOf not (arch:use some arch:Repository)
ASSERTED code:PersonEntity arch:use code:PersonRepository
ASSERTED code:PersonEntity a arch:Entity
ASSERTED code:PersonRepository a arch:Repository
```

Listing 7.3 shows an example of a violation that is an *absence* of the rule "`Every` *Repository* `must` *use* `an` *Entity*". In this example, the relation *use* between individuals of the concepts *Repository* and *Entity* is missing. No individual can be found, represented by the variable `x`, that can be inferred to be a member of the concept *Entity*. In this case, the violation can be described as $v = (PersonRepository, use, Entity, absence)$. The object of the violation refers to the concept *Entity* from the architecture ontology.

Listing 7.3: *Exemplary proof tree of the must rule type.*

```
VIOLATED arch:Repository rdfs:subClassOf (arch:use some arch:Entity)
ASSERTED code:PersonRepository a arch:Repository
NOT_INFERRED code:PersonRepository arch:use x
            x a arch:Entity
```

7.5. Ontology-Based Preservation of Architecture Conformance Checking Results

The conformance checking results revealed by the reasoner should be preserved, e.g., in order to be used for subsequent code reviews. The main idea is to enrich the knowledge base with the conformance checking results. This means, individuals responsible for violations and the violated architecture rules are connected with the results. In this way, the implemented architecture of a specific version of the software, the architecture rules, and the corresponding results are unified in one knowledge base. For this, the results also need to be stored in an ontology-based representation. In the following, the architecture conformance ontology is presented and how the results of the conformance check populate the ontology.

7.5.1. Architecture Conformance Check Ontology

In order to store the results in the database, the information about a conformance check needs to be represented as an ontology. This encompasses the following information:

1. general information about the conformance check, for example the date and time of the validation,
2. the architecture rules that are validated in the conformance check,
3. the resulting architecture violations,
4. the code entities that cause the violations,
5. the code version that is validated.

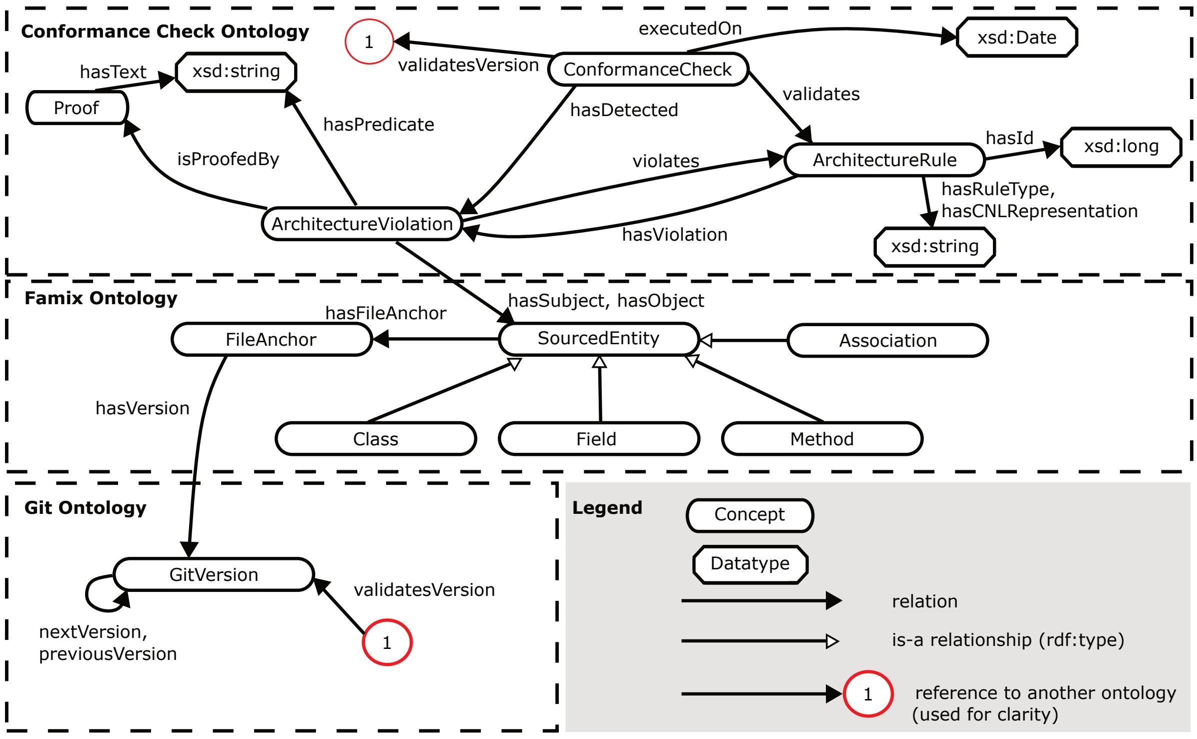


Figure 7.11.: *Conformance checking ontology and its integration with the FAMIX ontology and the Git ontology.*

In order to capture information 1-3, a new ontology is designed. The other information (4 and 5) can be reused from the FAMIX, Maven and the Git ontology. Figure 7.11 depicts the conformance check ontology and how it is related with the FAMIX and the Git ontology. It provides a unified integration of conformance checking results, the source code, and the source code history.

The concept ConformanceCheck represents the main concept to represent a conformance check. A ConformanceCheck validates a selected set of ArchitectureRules. An ArchitectureRule is identified by an ID (hasId) and is associated with its corresponding *ArchCNL* representation (hasCNLRepresentation) and its rule type (hasRuleType). ArchitectureViolation stands for architecture violations detected during a conformance check. An ArchitectureViolation captures which entities from the source code model are involved in the violation.

Individuals of ArchitectureViolation are connected with the violating elements from the code models via the hasSubject, hasObject, and hasPredicate properties corresponding to the subject, object, and predicate values of a violation as defined in Definition 7.4.2. In Figure 7.11 it is illustrated, how the conformance check ontology is connected with the FAMIX ontology. In this case, an ArchitectureViolation is linked with a SourcedEntity, which may be a class, a method, an attribute etc., via the hasSubject and the hasObject property.

Individuals of ArchitectureViolation are connected with individuals of the ArchitectureRule concept. In this way, information is captured about the entities, e.g. classes, methods, fields, responsible for violating a specific rule.

An ArchitectureViolation is described by a Proof. This is a detailed explanation given by the reasoner showing which axioms have been violated and why. A Proof has a

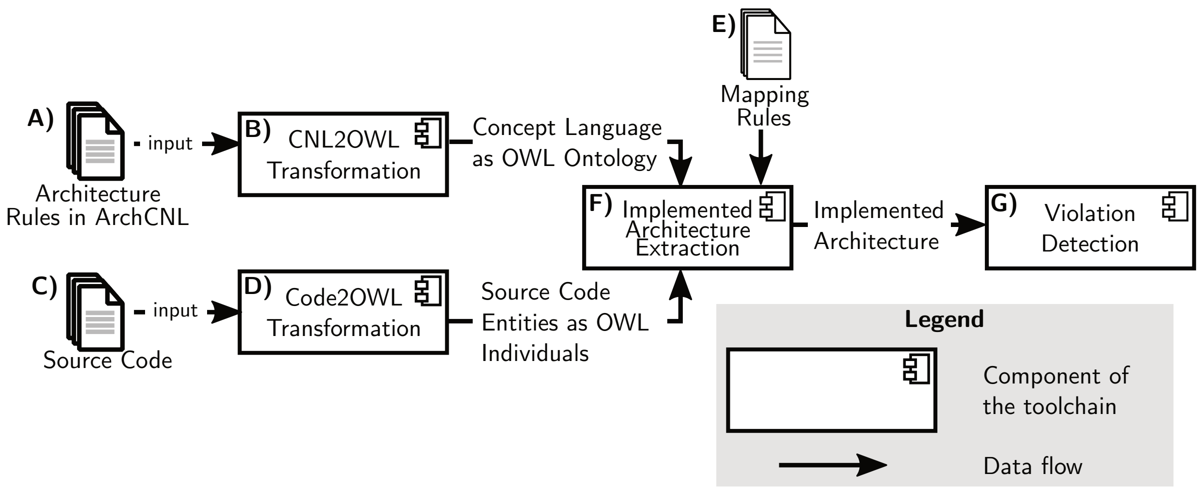

A)
Architecture Rules in ArchCNL
input
B) CNL2OWL Transformation
Concept Language as OWL Ontology
E)
Mapping Rules
F) Implemented Architecture Extraction
Implemented Architecture
G) Violation Detection
C)
Source Code
input
D) Code2OWL Transformation
Source Code Entities as OWL Individuals
Legend
Component of the toolchain
Data flow

Figure 7.12.: *Main components of the toolchain.*

description in a string format. This is captured by the relation hasText that connects a Proof individual with a string. The description of the proof follows the string representation of the proof tree as shown in Listing 7.1 and Listing 7.2.

The conformance check and FAMIX ontology are integrated with the Git ontology in order to store the conformance check results of different code versions. Individuals of SourcedEntity are associated with a GitVersion (a concept from the Git ontology) that is given by the version control system in which they are managed. This is necessary in order to store the version in which a code entity caused a violation. In this way, the evolution of architecture violations can be captured, e.g. by reviewing architecture violations from previous conformance checks and comparing the results of different versions.

The results are automatically stored in the knowledge base. As described in Section 7.4, a proof tree of an inconsistency represents concrete information about the architecture violation, i.e., which individuals cause the violation of the rule. Basically, the knowledge base is extended with additional facts that store these results. These facts are connected with existing ones that are part of the conformance check. In this way, a unified view on the conformance checking results, the implemented architecture, and the architecture ontology is achieved. The concrete procedure is described in Chapter F.

In a subsequent step, conformance checking results from previous checks can be retrieved by the software architect, e.g., for the purpose of documentation or architecture reviews. More precisely, the software architect writes SPARQL queries to retrieve the required information of a specific conformance check. Section F.1 describes some exemplary SPARQL queries.

7.6. Automating the Conformance Checking Process

As depicted in Figure 7.12, the approach is implemented in a tool chain that supports all the necessary steps for architecture conformance checking [KN16]: specification of architecture rules with *ArchCNL* (**A** and **B**), fact extraction from source code (**C** and **D**), definition of mapping rules (**E**) that are used to extract the implemented architecture (**F**), and analyzing the implemented architecture for violations (**G**) using reasoning services of the knowledge base [sta]. An open-source version of the tool chain is provided [git19].

The tool prototype supports the automatic transformation of 1) Java source code to the FAMIX ontology, 2) Maven POM files to the Maven ontology, and 3) Git history to the Git ontology.

Architecture rules can be documented inside a plain text file. Currently, they can be integrated in a text file that follows the asciidoc format [asc]. This has the advantage that the rules can be easily managed in a version control system. Moreover, a lot of asciidoc templates for architecture documentation exist - such as for arc42 [arca] or for ADR templates [adr] - so that architecture documentations can be enriched with architecture rule formalizations. The architecture rule documentation is automatically converted into integrity constraints.

Based on the mapping rules specified by the architects or developers, the implemented architecture is extracted by using a reasoner provided by Apache Jena [jen]. The implemented architecture is imported to the knowledge base (Stardog Knowledge Graph Platform [sta]) together with the architecture rules (represented as OWL integrity constraints). The tool prototype produces architecture violation reports that are again documented in asciidoc. This report depicts the respective violations for each architecture rule and an explanation why an architecture rule has been violated by referencing the corresponding part of the source code that violates the rule.

The grammar of *ArchCNL* is defined using Xtext [xteb]. The transformation from *ArchCNL* sentences to OWL axioms is performed with Xtend [xtea].

The reasoning process is performed by the reasoner implemented in Stardog. It is used since it supports the validation of integrity constraints, i.e., OWL axioms that are evaluated using the CWA. Stardog also provides as a database which unifies the architecture ontology, the code model (representing Java code, Maven build files, or a Git history), the individuals representing the implemented architecture, and the conformance checking results.

7.7. Conclusion

In this chapter, *ArchCNLCheck* has been presented as a novel approach for architecture conformance checking. The process consists of several steps including the 1) transformation of source code artifacts into an ontology-based representation, 2) rule-based extraction of the implemented architecture using SWRL rules, and 3) the calculation and ontology-based preservation of architecture violations.

For step 1) three source code artifact ontologies are proposed for representing object-oriented source code, e.g., Java, Maven build files, and Git histories. An upper ontology is provided so that more artifact types can be added that can be checked for conformance. Designing such ontologies is challenging. Ideally, ontologies are complete [Gó01], i.e., contain all concepts, relations, and axioms that are needed to describe and to reason on the domain. In the context of conformance checking, this means that the source code artifact ontology needs to cover all necessary information of the artifact type it represents. If not all concepts and relations important to the artifact type are covered, there is a high risk that crucial architecture violations cannot be detected.

The completeness of an ontology cannot be proved [Gó01]. However, it can be addressed and ensured to some extent in several ways. As one option to reduce the risk of incompleteness, well-established and well-documented meta models for representing artifacts can be used. In this thesis, the FAMIX meta model was used as a reference in order to derive the corresponding ontology and to represent object-oriented source code. The FAMIX meta model is known to be a comprehensive meta model for storing source code [DAB$^+$11]. By defining and following transformation rules it is ensured that all meta model constituents are correspondingly transformed to their ontology counterparts. Therefore, the ontology contains all information necessary to represent object-oriented source code with respect to the reference meta model.

In case of the other artifact types – e.g., Maven and Git – presented in this chapter, no meta model has been used as a reference. Instead, concrete artifacts have been investigated and the ontologies are a result of an iterative process. Evaluating ontologies and their completeness in particular has been a core research topic from the early stages of the Semantic Web resulting in a set of approaches that aim for evaluating the quality of ontologies [BGM05]. Existing evaluation methods for ontologies could be applied here in order to address the completeness of code ontologies. They could provide indicators to which extent the ontology is able to represent the domain of an artifact type. For example, the approach by [GF95] uses so-called *competency questions* which are a set of questions representing requirements that an ontology must be able to answer. In the context of architecture conformance checking these question could constitute specific types of architecture violations that can be detected based on a code ontology.

8. Evaluation

In the previous chapters the ontology-based enforcement approach is presented. The approach allows software architects and developers to a) define their project-specific architecture concept language using the *Architecture Controlled Natural Language (ArchCNL)* and b) validate the language against the implementation using *ArchCNLCheck*.

In this chapter, the evaluation of *ArchCNL* and *ArchCNLCheck* is presented. The aim of the evaluation is to assess 1) the flexibility and expressiveness of *ArchCNL* and thereby its ability to reflect the architecture concept language of a project and to formalize architecture rules, 2) the perceived applicability of *ArchCNL* in practice, and 3) the architecture violation detection quality of *ArchCNLCheck*.

Section 8.1 presents the goals and applied methods of the evaluation. In each subsequent section, the evaluation of the respective goals is presented: In Section 8.2, the results of an industrial case study are presented in which the expressiveness and flexibility of *ArchCNL* is evaluated. Section 8.3 presents the results of a focus group that assesses the perceived applicability of *ArchCNL*. Finally, Section 8.4 presents the results of applying *ArchCNLCheck* on two open-source systems for evaluating the architecture detection quality.

Parts of the evaluation are published in **[Sch18a]**, **[Sch19a]**, **[Sch19b]**, and **[Sch19c]**.

8.1. Evaluation Goals and Evaluation Methods

In this section, the goals of the evaluation are presented. Additionally, for each evaluation goal the employed methodology is described. Table 8.1 depicts the evaluation goals and the corresponding methodologies that have been applied.

8.1.1. Flexibility and Expressiveness

As described in Chapter 5, *ArchCNL* aims to be more expressive and flexible in terms of architecture concepts and relations that can be formalized. Consequently, it is hypothesized that it is able to formalize a diverse set of architecture rules that are relevant for validation. An industrial case study is conducted to assess in how far *ArchCNL* is able to formalize architecture rules from practice. For this, examples of architecture rules are collected from three industrial projects and categorized according to their characteristics. These examples are then formalized with *ArchCNL*. Based on interviews with the software architect of the respective project, it is assessed to which extent the formalization reflects the original intention of the rule. Additionally, the expressiveness of the approach is compared with existing approaches. In this way, the limitations and drawbacks of existing approaches are emphasized.

Table 8.1.: *Criteria, the corresponding methods to validate them, the corresponding section where the evaluation is described, and the respective thesis goal that is referred by the evaluation.*

Evaluation Goal	Method	Section	Thesis Goal
Flexibility and Expressiveness	Industrial case studies	Section 8.2	**G2**
Applicability	Focus Group	Section 8.3	**G2**
Architecture Detection Quality	Case studies	Section 8.4	**G3**

8.1.2. Applicability

One goal of the thesis is to provide an approach that allows for a more understandable formalization of a project-specific language and architecture rules, respectively. *ArchCNL* provides such an understandable and usable way to specify the architecture concept language and the corresponding architecture rules. In this part of the evaluation, it is evaluated whether the approach is indeed perceived as a useful means to document and formalize architecture rules by qualitatively and quantitatively assessing the applicability of *ArchCNL*. As recommended by other authors who evaluated the applicability of CNLs, applicability is evaluated according to the three aspects *understandability* [Kuh13], *usability* [WPT14] [KB14], and *naturalness* [Kuh14]. For this, a focus group with 12 developers guided by a survey has been conducted.

8.1.3. Architecture Violation Detection Quality

The architecture violation detection quality of the ontology-based conformance checking approach is evaluated using the two open-source systems TEAMMATES [tea] and JabRef [jab]. Architecture rules are formalized with *ArchCNL* and validated with the tool chain that implements *ArchCNLCheck* (see Chapter 7.6). The aim of this part of the evaluation is to assess to which extent the approach is able to detect relevant architecture violations. For each open-source system, a collection of approved architecture violations exists that is used as a *ground truth*. The detected violations are compared with the results contained in the ground truth and the measures precision and recall [Tin10] are used to assess the architecture violation detection quality.

8.2. Flexibility and Expressiveness

In this section, the flexibility and expressiveness of the approach is evaluated based on three industrial projects. Architecture rules from these projects are collected, categorized and formalized with *ArchCNL*. This section is structured as follows: First, the objectives and the research questions of the study are presented. Second, the industrial projects and their characteristics are described and how data is collected from these projects and how the data is analyzed. The Section 8.2.4, Section 8.2.5, and Section 8.2.6 present the results of the analysis and provide answers to the research questions. Section 8.2.7 assesses to which extent existing approaches are able to formalize these rules from practice.

8.2.1. Objective and Research Questions

The aim of the study is to validate the flexibility of the approach by applying it on architecture rules identified in industrial projects. Therefore the following research questions (RQs) are investigated:

RQ1: What kind of architecture rules exist in industrial projects?

With this question it is investigated which different kinds of rules exist in projects and which characteristics they have.

RQ2: In how far is the approach able to formalize those architecture rules from practice?

The goal of this RQ is to investigate if and to which degree the rules identified in RQ1 can be formalized with *ArchCNL*. Furthermore, the characteristics of rules that cannot be formalized are determined.

8.2.2. Units of Analysis

The approach is analyzed based on three industrial projects:

- *Project 1* is a framework for providing static code analysis as a service. The framework has been developed for analyzing service-based software systems in the finance domain. The framework has a size of 26 thousand lines of code (kLoC) and has been developed over three years by three developers. The system is developed with Java and Spring Boot.
- *Project 2* is a software framework for domain logic extraction and documentation generation with focus on software legacy systems. The framework has been used for extracting knowledge from engineering software as well as business rules from legacy software developed by companies from finance and insurance domains. The framework was developed in Java over a period of 7 years by 7 developers and has a size of 490 kLoC.
- *Project 3* is a software for the programming of industrial welding robots by end-users. The software is comprised of a visual DSL, a dedicated graphical editor, and a code generation framework. The software is developed in C# and WPF and has 120 kLoC that were contributed by 4 different developers over a period of three years.

8.2.3. Data Collection and Analysis

The study was conducted in two phases which are described in the following.

Phase 1) Rule Collection and Categorization: This phase aims for answering **RQ1**. In a first step, software architects of the three projects are asked to collect architecture rules in their projects. These rules were provided by the architects as part of the architecture documentation of these projects. The rules are described (informally) in natural language (English). In order to answer **RQ1**, artifact analysis is applied as a technique for characterizing the architecture rules. Open coding and the constant comparison method [Gla78] has been applied for the analysis. Architecture rules are labeled with codes that appropriately classify the characteristics of the rule. The codes are compared with each other within the document and with codes from the architecture documentation of the other projects. If appropriate, similar codes were merged to more high-level concepts. Those concepts constitute the rule categories. For the categorization, no predefined codes or categories have been used. The categories have emerged from the data by applying the open coding method. Two researchers were involved in the process. They have analyzed the artifacts independently and developed categories using the before mentioned methods. After that, they have compared their categorization. The categorization was repeatedly discussed and restructured in an iterative process.

Phase 2) Rule Formalization and Interviews: This phase aims for answering **RQ2**. In this phase, architecture rules provided by the software architects are formalized with *ArchCNL* (see Chapter 6). After this, the architects were interviewed. Each architect has been interviewed once. Each interview has been conducted by the same interviewer (the author of this thesis). In these interviews, the architects were asked to assess the formalization given in *ArchCNL* of the architecture rules they have provided, i.e., each architect evaluates the formalization of the rules of his project. During the interviews, the formalization of each rule was presented by the interviewer on a sheet of paper containing the natural language description of the rule and its corresponding formalization in *ArchCNL*. The rules have been processed successively. For each rule, the architects were asked by the interviewer to judge whether the representation of the rules in *ArchCNL* still reflects the intents of the original rules. This aims for answering **RQ2**, i.e., in how far *ArchCNL* is able to formalize architecture rules found in practice. The interview was performed following a interview guide with closed and open-ended questions according to the guidelines of [Cha14]. The interview guide can be found in the appendix (see Chapter C). It contains questions to capture the background of the participant, questions that aim to assess the suitability of the formalization of the architecture rules, and open-ended questions regarding the general impression on the approach. In total, the interviews took 5.3 hours, where each interview took 1.7 hours in average. Table 8.2 shows the demographics of the study capturing the experience of each participant. The interviews have been conducted in October 2018.

For further analyses, the interviews were transcribed word-by-word. The transcripts have been browsed for passages relevant for the research questions. First, the background information relevant for demographics is extracted. After that, data relevant for **RQ2** is analysed. For each rule, it is assessed based on the answers given by the participants whether its formalization was perceived appropriate, i. e., the original intention was preserved. The discussion about a concrete formalization provides qualitative data which was analyzed using open coding and the constant comparison method [Gla78]. With this method, reasons for inappropriate formalization and other aspects could be revealed.

8.2.4. RQ1: What kind of architecture rules exist in industrial projects?

In total, 56 architecture rules have been found, where project 1 contains 18 architecture rules, project 2 contains 8 architecture rules, and project 3 contains 30 architecture rules. Hence, based on the analysis, the following types of architecture rules have been revealed:

Design Rules: Rules that enforce that system parts are realized in a prescribed way.

Functional Requirements: Rules that define specific program functionality.

Table 8.2.: *Demographics of each software architect (SA) of the respective project investigated in this study.*

SA	Project	Years of Experience	Years working in project	Domain	Technology Stack
1	Project 1	12	3	Banking, Automatization	Java
2	Project 2	20	7	Software Development Tools	Java
3	Project 3	12	3	Banking, Automatization, Domain Specific Languages	C#, Java

Static Dependency Rules: Rules defining how system parts are allowed/not allowed to statically depend on each other.

Coding Guidelines: Rules that ensure that functionality is implemented in a unified way. E.g., that REST APIs are documented with a particular annotation and map exceptions to HTTP error codes.

Use of Technology: Rules that enforce the use of particular technologies like programming languages and frameworks.

Non-Functional Requirements: Rules that prescribe quality goals for the whole system or system parts.

Code Quality: Rules that aim to detect code smells. For example, such rules define that a class should not define more than a specific number of public methods in order to prevent the bad smell *God Class* [Fow99].

Figure 8.1 shows how often a category was found in each project. Based on the codes and the resulting categories, it can be observed that architecture rules refer to different levels of abstraction, reaching from rather high-level rules to rules defined on source code level. As can be seen, the *design rule* category is the most strongly represented category in the case studies. This category refers to architecture rules that are defined on a high level of abstraction. This type of rule, for example, defines which architecture patterns [BMR+96] must be used, enforces specific architecture design principles (e.g., separation of concerns), specifies parts that must be extended or should not be changed when new functionality is added, or defines which operations must be provided by dedicated interfaces.

Functional requirements were also defined frequently as a part of architecture documentation, especially in project 3.

The third strongly represented category is the *static dependency rule* category. This rule category is well supported by existing approaches and tools such as [TV09, PKvdWB14, CLN15].

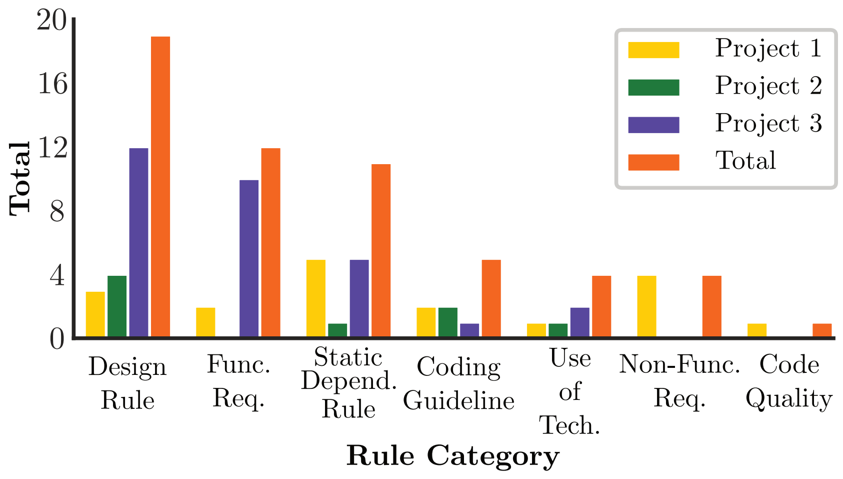


Figure 8.1.: *Total number of rule categories found in the projects.*

Evaluation: Answer to RQ1

Based on the results, it can be observed that architecture rules with different characteristics exist in industrial projects. Moreover, it was found that only **20%** of the rules can be considered **static dependency rules**. This rule category is well supported by state-of-the-art conformance checking tools. However, it was also found that a significant proportion of the rule categories are not supported by these tools, e.g., **design rules** that make up **40%** of the architecture rules discovered.

8.2.5. RQ2: Is the approach able to formalize those architecture rules from practice?

The amount of rules that can be successfully formalized with the approach is further analyzed. A rule is classified as “successfully formalized” when the corresponding software architect of the project approved during the interview that the original intention of the rule was appropriately reflected in the formalization.

For each project, Figure 8.2 visualizes the amount of rules that can be formalized and that cannot be formalized with *ArchCNL* for each category in a bar plot. Additionally, the aggregated number of the formalized rules and rules that could not be formalized is shown for each project. The amount of rules that could not be formalized is further divided into the amount of rules that are not supported by the formalism (“Not supported”) and the amount of rules that are not useful to be formalized (“Not useful”).

Generally, it can be observed that the amount of rules that have been successfully formalized is significantly higher than the amount of rules that could not be formalized. This means that most architecture rules could be formalized. Table 8.3 depicts an excerpt of architecture rules taken from the industrial projects and their corresponding formalization in *ArchCNL*. The rules can be formalized without loss of their original intention, since architecture concepts and relations used in the natural language description can be directly represented in the formalization. Some rules in *ArchCNL* even directly represent the natural language description, see for example rule 1 in Table 8.3. The approach therefore provides a great flexibility in terms of architecture rule formalization.

Table 8.3.: *Rule examples written in natural language (**NL**) for each rule category found in the projects and their corresponding formalization using the ArchCNL notation (**CNL**).*

1	**NL**	*Exceptions raised by business logic services must be mapped to corresponding HTTP error codes.*
	CNL	`Every BusinessLogicException must map-to a HTTPErrorCode.`
	Category	Coding Guideline
2	**NL**	*The REST API must not make any direct database access by using functionality provided by the repository or the repository.cache packages.*
	CNL	`No RESTAPI can access a Repository or can access a RepositoryCache.`
	Category	Static Dependency Rule
3	**NL**	*Avoid large controller classes with too many methods.*
	CNL	`Every RESTController can define at-most X Methods.`
	Category	Code Quality
4	**NL**	*For each REST API, a dedicated feign client should be provided in a dedicated client module.*
	CNL	`(1) Every ServiceModule must provide a FeignClient.` `(2) Every RESTAPI must map-to a FeignClient.`
	Category	Design Rule
5	**NL**	*Database caches must be cleared whenever a new module is parsed.*
	CNL	`Every ModuleParseOperation must clear a DatabaseCache.`
	Category	Functional Requirement
6	**NL**	*It must be ensured that database caches do not grow endless in memory during runtime, e.g. by the use of "least recently used maps".*
	CNL	`Every Collection that (is-used-by a DatabaseCache) must be a LeastRecentlyUsedMap.`
	Category	Non-Functional Requirement
7	**NL**	*If no open-source parser is available for your favourite language, use CoCo/R for the generation of scanners and parsers (in the Language frontend). Please avoid using AntLR.*
	CNL	`(1) Every LanguageFrontend must use a CoCoRLibrary.` `(2) Nothing can use an ANTLRLibrary.`
	Category	Use of Technology

However, it can be observed that a few architecture rules cannot be formalized. Some natural language descriptions contain constructs that cannot be easily transformed to *ArchCNL*, since the underlying formalism does not support the constructs. For example, some rules contain temporal constraints, e. g., rule 5 in Table 8.3. Although a formalization is provided, the rule is not properly formalized. As can be seen, the temporal constraint is not reflected in the formalization, since temporal constraints are not supported by the description logic formalism. Therefore, the formalization could not directly reflect the original intention. Consequently, the rule is classified as “not supported”. That is why, additional language features are needed or the underlying formalism needs to be extended in order to support the formalization of those rules. In [Jas20], the ArchCNL has been adopted with language elements for representing temporal constructs. In this way, the ArchCNL can be used to enforce security-related architecture rules in the implementation.

Additionally, as stated by the interview participants, it was not considered reasonable and

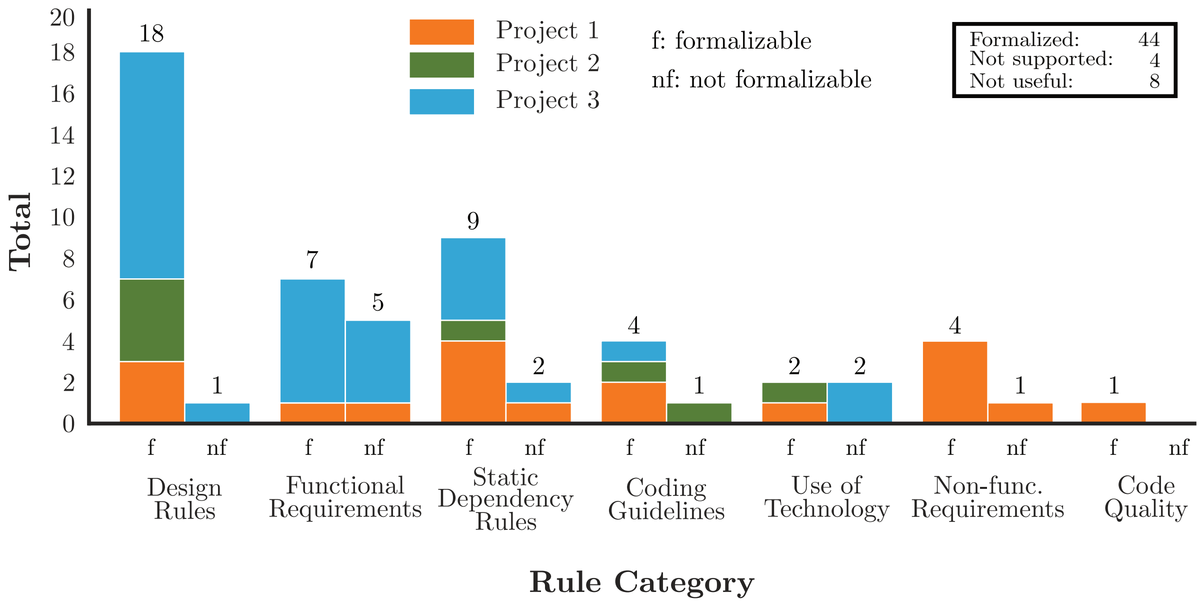


Figure 8.2.: *Amount of rules in each project – categorized by the rule categories – that can be formalized and that cannot be formalized with the approach.*

necessary to formalize all of the architecture rules. Based on the qualitative analysis, several reasons are identified for this:

1. Violating this rule would be obvious. Those kind of rules usually prescribe the use of a particular technology or a programming language: *"...I am not sure if it is reasonable to formalize or validate the rules in the context of this project or to validate the rules that I have described here. Those rules are very generic and prescribe very generic requirements, like .NET, WPF and so on. The value of formalizing and validating them would be very minimal, because it is guaranteed that it is used anyway..."* (project 3).

2. Rule conformance is enforced by the framework used in the project. Consequently, it is simply not possible for a developer using the framework to break the rule. For example, in project 3, the system implementation is based on a framework that enforces a specific way how some parts of the software system must be implemented: *"...UI components can only be integrated via the main application. This is already ensured by the application itself. During the development of the framework, the validation of this rule would be difficult. And actually, this is not possible..."* (project 3).

3. Rules are classified as "non-goals". For example, the rule *"An implementation in common packages needs not be thread-safe."* (project 2) represents such a non-goal. First of all, such a rule type is not supported by *ArchCNL*. The most similar type would be the negation rule type: `No CommonPackageImplementation can be a ThreadSafeImplementation.` However, this formalization expresses a different intention, namely, it forbids thread-safeness. The original intention of the rule is to state that thread-safeness is not a necessary requirement for the mentioned packages.

Nevertheless, it could still be advantageous to formalize architecture rules although they are not used for architecture conformance checking as it will be discussed in the next section.

8.2.6. Further Observations

In general, participants find that the approach is applicable for formalizing architecture rules. Based on the analysis of the transcripts, it is found that participants appreciate the flexibility and expressiveness of the approach. This was also mentioned by an architect: *"I like the openness of the approach. On the one side you need to define the glossary, on the other side you have the possibility to extend it and to create a project-specific language. This is what I like the most..."* (project 3).

The unambiguity of architecture rules formalized with the approach is also seen as an advantage. Additionally, participants find that the language is easy to learn, since it has a manageable grammar. The approach also greatly supports architects and developers even without validating the rules automatically: *"I think that this formalization definitely has a value independently of whether the rules are automatically validated or not..."* (project 2). This involves different aspects: Firstly, participants think that *ArchCNL* is an appropriate means to be used in the team in order to find a consensus about architecture concepts and relations used in a project. Secondly, applying the approach helps to improve the quality of architecture rules, since it supports to clarify and clearly define the architecture concept language used in a project. Thirdly, despite the fact that some rule formalizations cannot be verified against the source code, it could still be advantageous to integrate them into the architecture documentation for architecture knowledge preservation, for unambiguous architecture documentation, and for sustaining the architecture concept language.

During the interviews, the software architects sometimes perceived that the formalization did not appropriately reflect the original intention of the architecture rule. Two different reasons have been identified: First, in some cases the formalization did not use the correct vocabulary that was intuitively understandable for the architect. For example, more than one term for identical architecture concepts and relations were chosen by the software architect. After applying those changes on the vocabulary, the architect perceived that the original intention of the rule was appropriately reflected. The second reason is that the natural language specification was ambiguous or incomplete. The software architects recognized that the formulation in natural language was not precise enough or the terms were used inconsistently. In those cases, the natural language description was revised as well as formalization in *ArchCNL*. As a result, the original intention was reflected more appropriately in both descriptions (*ArchCNL* description and natural language).

The fact that architecture concepts and relations need to be defined explicitly as part of the rule formalization process, enforces to think more concretely about chosen terms for concepts and relations and to use those terms consistently across the architecture documentation. This means that the approach aids to unambiguously define architecture concepts and relations and the corresponding architecture rules. That also emphasizes how the approach can greatly help to find weaknesses in the architecture documentation regarding the inconsistent use of the vocabulary. This was also realized by one of the participants during the discussion: *"what I realize here is that the approach forces me to define the concepts more explicitly. In natural language you often use synonyms, like Controller class or REST controller. One reason could be that, during writing the rule, you do not realize that you do not use the same word. However, those words may have a different meaning. That is why, concepts should be named clearly, actually how it should be done in a software documentation."* (project 1).

Evaluation: Answer to RQ2

Nearly **80%** of the architecture rules found in the industrial projects have been successfully formalized. This means that *ArchCNL* is able to preserve the original intention of the majority of the architecture rules. Compared to existing approaches for architecture conformance checking, the approach is not restricted to static dependency rules and is able to formalize architecture rules from other categories. The proportion of rules that could not be formalized **(20%)** is acceptable, since this set mostly contains rules which are considered to be not useful to be formalized ($\approx$ **67%** of rules that could not be formalized are considered as to be not useful for formalization).

8.2.7. Expressiveness of Existing Approaches

The empirical study has shown that *ArchCNL* is sufficiently expressive and flexible to formalize architecture rules in practice. Since ontologies are not restricted to existing concepts and relations, terms for architecture concepts and relations used in natural language descriptions can be used as first class entities in the rule formalization in *ArchCNL*. Due to this, the approach provides a higher flexibility than existing approaches providing strict meta models for architecture rule formalization. In order to illustrate this, three exemplary, publicly available tools have been chosen and applied to a set of architecture rules from project 1 of the industrial case study (see Section 8.2). Each tool defines a specific meta model containing predefined architecture concepts and relations the intended software architecture can be described with. An overview on the architecture concepts and relations of each tool is depicted in Table 8.4. The following tools are used for the comparison:

Dependency Constraint Language (DCL): DCL [TV09] is a textual DSL that allows for the specification of module dependency rules. The meta model of DCL provides the architecture concept *module*. Modules aggregate a set of classes. Dependencies between modules are restricted by formalizing rules on how modules are allowed to be related with each other. DCL supports the architecture relations *use, access, depend, throw, handle, create* to describe dependencies between modules. These relations are mapped to their corresponding code-level relation in Java code. It is not possible to define additional, project-specific architecture concepts and relations in DCL.

Dictō: Dictō [CLN15] is a textual DSL for defining rules on quality requirements. It aims for unifying a set of diverse tools quality attributes can be measured and validated with. Dictō provides a unified, usable interface for those tools. The meta model of Dictō defines so-called *entities* for which the constraints are described. These entities are predefined by the syntax of Dictō. Entities can be files, Java code elements (classes, methods, packages), or XML elements etc. Dictō prescribes a set of architecture relations (see Table 8.4), called *predicates*. Each predicate maps to a specific wrapper which delegates the validation of the specified rule to the corresponding tool. For example, the predicate *depend on* is assigned to the tool Moose [moo] which is able to validate this predicate, i.e., to validate whether there is a *depend on* relationship between two specified entities.

HUSACCT: HUSACCT allows for defining semantically rich modular architectures [PKvdWB14]. The tool provides a graphic-based view to define the intended software architecture in

Table 8.4.: *Architecture conformance checking approaches used for the comparison and their supported architecture concepts and relations for defining the intended architecture.*

Approach	**Architecture Concepts**	**Architecture Relations**
DCL	module	access, declare, handle, create, extend, implement, derive, throw, use-annotation, depend
Dictō	entity (class, file, package, method ...)	depend on, invoke, have annotation, have method, implement interface, have method parameter, throw, catch, contain code clones, contains cycles, lead to deadlock
HUSACCT	layer, component, subsystem, interface, external component	use, implement, extend

terms of specific module types. These module types are *layer*, *subsystem*, *component*, *external system*, and *interface* and are predefined by the meta model of HUSACCT. Each module type is associated with a set of predefined rule types. For example, modules of type *layer* are not allowed to use modules that are upper layers. In contrast to DCL, the relation *use* is an aggregation of code-level relations, e.g., access, invoke, returns etc. This mapping is predefined by the tool and cannot be changed by the software architect in the rule definition. The intended architecture in HUSACCT is defined with a graphical model, whereas the architecture rules for each module are given in a text-based representation. In the exemplary formalizations (as in Figure 8.3), the text-based representation of rules is used.

There are, of course, many other ACC tools available that could be used for comparison. However, this current selection of tools should suffice to act as representative examples in order to show the limitations and challenges when formalizing architecture rules from practice. This study is considered as a quasi-experiment as defined by Wohlin [WRH$^+$12] since the selection of investigated tools and architecture rules is not randomized.

Figure 8.3 depicts the exemplary architecture rules taken from project 1 for the comparison and to which extent they can be formalized with the tools presented above. Each rule has been formalized with each approach. For each rule, terms indicating architecture concepts and relations are extracted that are eventually mapped to the architecture rule formalization.

Rule 1) is a representative example of a static dependency rule. It can be seen that this rule can be quite well formalized with the approaches. This means that the concepts and relations used in the natural language description are appropriately represented by the architecture concepts and relations provided by the tools, i.e., *module* in DCL, *entity* in Dictō, and *subsystem* in HUSACCT. For example, in DCL, the concepts *RESTAPI* and *Business Logic Service* are mapped to *modules* and the relation *delegate to* can be interpreted as a *depend* relation.

Although for nearly all rules a formalization is provided, not all formalizations are appropriate due to several reasons:

1) The REST API (classes with @RestController annotation) must only delegate to Business logic services (services located in the services package)

concepts: RESTAPI, Business Logic Service **relation**: delegate to

DCL	`RESTAPI can-depend-only BusinessLogicService`
Dictō	`RESTAPI can only depend on BusinessLogicService`
HUSACCT	`RESTAPI is only allowed to use BusinessLogicService`
ArchCNL	`Every RESTAPI can only delegate-to a BusinessLogicService.`

2) Exceptions raised by business logic services must be mapped to corresponding HTTP error codes.

concepts: Business Logic Service Exception, HTTP Error Code **relation**: map to

DCL	`BusinessLogicServiceException must-useannotation @ResponseStatus`
Dictō	`BusinessLogicServiceException must have annotation @ResponseStatus`
HUSACCT	`BusinessLogicServiceException must use springframework.[...].ResponseStatus`
ArchCNL	`Every BusinessLogicServiceException must map-to an HTTPErrorCode.`

3) Avoid large controller classes with too many methods.

concepts: Controller class, method **relation**: define (implicit)

DCL	- rule type not supported -
Dictō	- rule type not supported -
HUSACCT	- rule type not supported -
ArchCNL	`Every RESTController can define at-most X Methods.`

4) For each REST API, a dedicated feign client should be provided in a dedicated client module, i.e., each service module must provide a feign client and each REST controller must map to a feign client.

concepts: REST controller, Feign Client, Service module **relation**: provide, map to

DCL	`ServiceModule must-depend FeignClient`
	`RestController must-depend FeignClient`

Dictō	`ServiceModule must depend on FeignClient`
	`RestController must depend on FeignClient`

HUSACCT	`ServiceModule must use FeignClient`
	`RestController must use FeignClient`

ArchCNL	`Every ServiceModule must provide a FeignClient.`
	`Every RESTController must map-to a FeignClient.`

Figure 8.3.: *Exemplary rules taken from project 1 of the industrial case study used for comparing the expressiveness of existing approaches with ArchCNL. Additionally, the formalization in ArchCNL is given.*

Unclear semantics of predefined relations: As mentioned before, all tools are able to formalize static dependency rules as for example rule 1) in Figure 8.3. The relation *delegate to* has been interpreted as a *depend* (DCL), *depend on* (Dictō) or *use* (HUSACCT) relation. In all tools, these relations refer to static code-level dependencies. However, the tools define the semantics of this relation differently although they use similar terms for this relation. This means that the architecture relation is mapped differently to code-level relations by each tool. As a result, it may not be clear whether the semantics of the architecture relation matches the desired purpose of the project, i.e., whether this relation covers all relevant code-level dependencies needed to validate the conformance. That is why, *ArchCNL* is more appropriate, since the architect can clearly define what *use* or *depend* should mean in a specific project by defining the corresponding architecture-to-code-mapping.

Rule type not supported: None of the tools provide support for the cardinality rule type. Rule 3 in Figure 8.3 is an example of an architecture rule that requires this rule semantics. However, this architectural rule type is considered important in order to validate the complexity of code elements that implement an architecture concept. In contrast, the approach developed in the thesis supports the formalization of cardinalities by exploiting cardinality role restrictions of the description logics formalism (see Chapter 6).

Semantic mismatch: In this context, two concepts or two relations are semantically mismatched if they have no sense in common, i.e., the terms used to describe the concepts or relations have no common meaning. Rule 4) shows an example of a semantic mismatch. In this rule, the relation *provide* and *map to* are formalized as *depend* (DCL), *depend on* (Dictō), and *use* (HUSACCT) relations in the respective rule formalization. Those relations have been used, since there are no other relations defined by the tool languages that could better represent the intended meaning of the relations given in the natural language description. However, these relations provided by the tool's language is inappropriate due to the different meanings of the relations. For example, the use relation between modules (as applied in HUSACCT) can be defined as follows according to [CGB+10]:

Definition 8.2.1: use

A module M_1 uses module M_2 if M_1 depends on the presence of a correctly functioning M_2 in order to satisfy its own requirements.

The relation refers to static, aggregated code dependency relations between two modules. In the context of the investigated tools, the relations *use*, *depend* and *depend on* are considered as synonyms.

This is not the intended meaning of the relations *map to* and *provide* in the context of rule 4). For example, *map to* can be defined as:

Definition 8.2.2: map to

A REST controller maps to a feign client if it exhibits the properties of a feign client.

This means, predefined relations cannot appropriately match the intended meaning of the relations given in the natural language description resulting in a semantic mismatch. As a consequence, the architecture rule cannot be appropriately formalized. With *ArchCNL*, a new architecture relation can be defined that appropriately represents the actual meaning of the relation given in the natural language description of the architecture rule.

Strong assumptions on implementation of concepts and relations: The rule formalizations of DCL and Dictō of rule 2) make a strong assumption on how the architecture relation *map to* is actually implemented in the code. As a matter of fact, the mapping of an exception to an HTTP error code is solved by annotating exception classes that implement the *Business Logic Service Exception* concept with the `@ResponseStatus`[1] annotation in project 1. That is why, the `useannotation` (DCL) and `have annotation` (Dictō) relations can be used for the formalization. However, this implies a decreased reusability of architecture rules, since rule formalizations are tightly coupled with a concrete implementation of concepts and relations. In another project, where this rule also applies, the architecture-to-code-mapping could be realized differently. For example, *map to* could be implemented by extending a specific library class. However, since the presented tools only provide a predefined set of architecture relations which cannot be extended with user-defined ones, it is not possible to formalize the rule without assuming a concrete implementation. Using *ArchCNL*, terms for concepts and relations can be used that are independent of such assumptions. Since the architecture-to-code-mapping is exchangeable a specific implementation of architecture concepts and relations can be configured independently of the rule formalization.

8.2.8. Threats to Validity

In this part, the threats to validity [WRH+12] of the study are discussed.

Conclusion validity: To achieve the reliability of measures, each rule formalization that was subject of analysis was discussed with and verified by the software architects of the three projects during the interviews.

Internal validity: To address the selection threat, projects from different domains have been analyzed. The projects also used different technology stacks. To address the history threat, projects that were running for multiple years have been selected that provided stable architectures and well established architecture rules.

Construct validity: To ensure validity of the constructs, the research questions and systematically selected methods for data collection and analysis are clearly defined.

External validity: With the study, external validity is not claimed due to the number of analyzed industrial projects (3) which is too small to generalize the findings. This implies that more empirical studies are needed to generalize the results in the future. This means that the approach has to be used in more industrial projects in order to get better insights which kinds of architecture rules are used in industrial practice, which additional features

[1] `https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/ResponseStatus.html`

are required by the approach, and how the language needs to be designed to better reflect the actual needs of its users. Nevertheless, the preliminary findings already provide valuable input for future research and have helped to identify open research challenges.

8.3. Applicability

In this section, the perceived applicability of the approach is evaluated. Thus, this section aims for answering the following research question (RQ):

> **RQ: How do practitioners perceive the applicability of *ArchCNL* as a means for architecture analysis and documentation?**

Three different aspects of applicability are assessed, namely *understandability*, *usability*, and *naturalness*. First, understandability captures to which extent participants perceive that the intention of architecture rules formulated in *ArchCNL* is directly clear. Second, the aspect usability assesses the perceived ease of applying *ArchCNL* in order to formalize architecture rules. Third, an architecture description in *ArchCNL* should appear as natural as possible in order to be human-readable. Thus, participant rate how natural an architecture rule written in *ArchCNL* actually appears to them. The applicability is evaluated in a workshop guided by a survey with experienced practitioners.

8.3.1. Data Collection

A workshop with 12 experienced software developers has been conducted to evaluate the applicability of *ArchCNL* from a practitioner's point of view. During the workshop, the participants evaluated 26 different architecture rules of different complexity with regard to understandability, usability, and naturalness. The architecture rules were taken from various sources, e.g. literature (such as [WB14]), websites, or open source systems. The workshop was guided by a presentation and a questionnaire that had to be completed by each participant. In the workshop a motivation on the approach presented is presented first and the professional background of the participants is assessed in the survey. Then, each of the 26 rules is individually evaluated. First, the *ArchCNL*-based representation is presented, followed by an informal description of the same rule. After each rule, there was a short discussion where the participants could comment on the rule and ask questions. One of the workshop organizers took notes on these discussions. Finally, the participants were asked to answer questions about the general impression of the *ArchCNL* and there was a final discussion on the general applicability of *ArchCNL*.

8.3.2. Participants Background

The data for this case study was collected from employees at the Software Competence Center Hagenberg (SCCH)[2]. It is a research center with a focus on applied research, i.e., the application of novel software development practices and techniques in industry. All members (15) of the software engineering group have been asked for participation. 12 members participated in the workshop and 3 members further agreed on analyzing their projects. Most of the participants

[2]`https://www.scch.at/de/aktuelles`

have multiple years of experience of work in industrial projects in different domains. 10 of the participants had more than 10 years of professional experience. One participant had 1 year of experience, and one participant had 3 years of experience. 8 of the 12 participants had not used any formal methods in their career. 2 participants are experienced in Behavior-Driven Development (BDD) [Sma14] and model-based testing, 2 other participants are experienced in domain-specific modeling and model-driven architecture.

8.3.3. Data analysis

In the study, qualitative and quantitative data is collected. Answers to open-ended questions in the survey and notes on workshop discussions provide qualitative data. Qualitative data is analyzed using open coding and the constant comparison method [Gla78]. Artifacts were browsed and searched for passages relevant for the respective research questions. The procedure followed for open questions is similar to the data analysis as described in Section 8.2.3. The passages were labeled and later grouped into categories. Likert-scale survey-questions are assessed using quantitative analysis. The concrete assessments are described in the following sections.

8.3.4. Quantitative Analysis of Understandability, Usability, and Naturalness

The goal of the study is to investigate the applicability of the approach with respect to the aspects understandability, usability, and naturalness. For evaluating these aspects, the Likert scale entries of the survey answers are transformed to integers. This way, the three aspects understandability, usability, and naturalness are quantified by a score for each single architecture rule, where 1 is the best score and 5 is the worst score for an aspect.

Architecture rules formalized in *ArchCNL* have a varying *structural complexity.* The structural complexity of an architecture rule is quantified by counting the number of a) architecture concepts, b) architecture relations, c) relative clauses, d) coordinators, and e) variables in *ArchCNL.* Architecture rules can then be classified into four groups with increasing structural complexity:

- **Group 1** contains neither variables, relative clauses, nor coordinators.
- **Group 2** contains either one variable, one relative clause, or one coordinator.
- **Group 3** contains one of the possible combinations: (variables, relative clause), (variables, coordinators), (relative clause, coordinators).
- **Group 4** combines all the three types with arbitrary frequency: variables, relative clauses, coordinators.

A rule maps to exactly one group. For each rule, the mean rating of all workshop participants for each aspect is calculated. Figure 8.4 depicts the mean ratings of the three aspects understandability, usability, and naturalness for each architecture rule. The dashed line connects the mean ratings of each group and visualizes the trend of the rating with increasing structural complexity. There is a large variation of about four scale divisions for all groups and all aspects so that each rule should be considered individually for a detailed analysis. Nevertheless, general trends can be observed. As expected, all aspects more or less get poorer for increasing

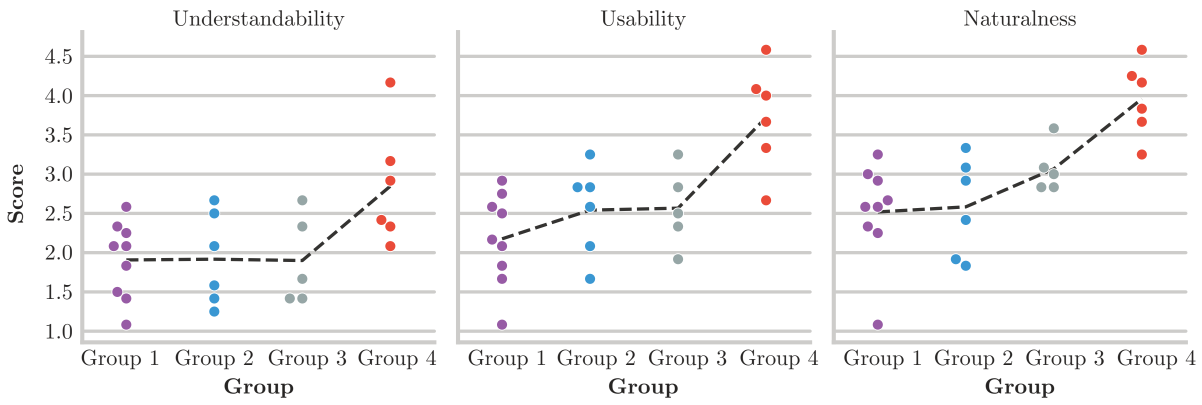


Figure 8.4.: *Mean rating for understandability, usability, and naturalness for each complexity group. Lower scores correspond to more desirable results.*

complexity. All groups are rated as rather understandable. However, it seems that structural complexity has a higher impact on the aspects usability and naturalness. Group 1 and group 3 show a rather good usability whereas group 4 is not considered usable anymore (average score ≈ 3.72) Group 4 also lacks naturalness (average score ≈ 3.95) whereas group 1 and group 3 show a rather mediocre rating of this aspect.

8.3.5. Quantitative Analysis: Overall Evaluation of ArchCNL

In the following, the results of the third part of the survey are presented. This part covers questions regarding the overall impressions of the participants on the approach after the architecture rules and their formalization have been presented. The answers of each question are aggregated into histograms in Figure 8.5. For each question, the average rating over all participants is depicted. Lower scores correspond to a higher approval to the statements given in the survey.

Participants perceived *ArchCNL* suitable for architecture rule documentation in general (Figure 8.5 (1)). Figure 8.5 (2) captures whether participants perceive that architecture rules represented in *ArchCNL* can be well understood by team members. In average, this was evaluated slightly poorer than neutral by the participants (average score = 3.25). There is a positive feedback on the flexibility of the approach, i.e., its ability to represent the architecture language used in a project, see Figure 8.5 (3). They also find that the language can be learned in short time, see Figure 8.5 (4). Participants also find that the approach is able to document rules that are relevant for their project (Figure 8.5 (5)). Generally, the participants think that *ArchCNL* could support developers on knowing important architecture rules (Figure 8.5 (6)) in the project and following them (Figure 8.5 (7)).

8.3.6. Qualitative Analysis and Answers to the Research Questions

In this section, the data of the quantitative analysis is interpreted. The results from qualitative analyses of the answers given in the open-ended questions in the survey and the discussion notes that were written during the workshop are presented in order to provide answers to the research question.

In general, most architecture rules presented in the workshop have a good rating for the

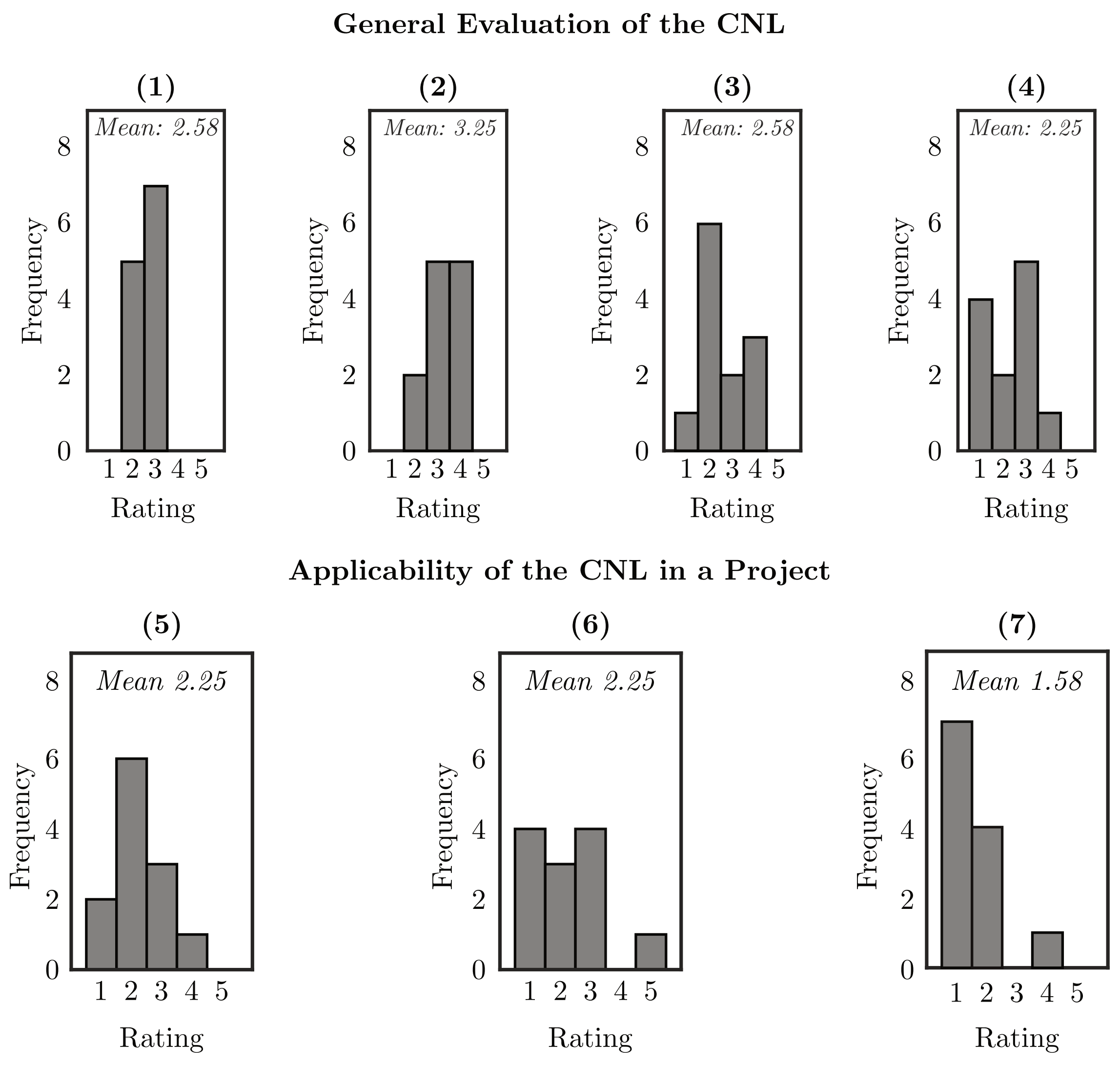


Figure 8.5.: *General evaluation and applicability of ArchCNL captured by the survey questions. Lower scores correspond to more desirable results.* ***(1)*** *ArchCNL is well suited to be used for architecture rule documentation.* ***(2)*** *The architecture rules documented in ArchCNL can be well understood by all team members.* ***(3)*** *It is possible to formulate architecture rules in ArchCNL, so that they are similar to the language used in the project.* ***(4)*** *I can learn ArchCNL in a short time.,* ***(5)*** *It is possible to document a lot of rules that are relevant for my project with ArchCNL.* ***(6)*** *The ArchCNL-based documentation would support developers and architects to know the most important architecture rules in the project.* ***(7)*** *The automatic validation of ArchCNL-based rules would support developers and architects to follow the most important architecture rules in the project.*

aspects understandability, usability, and naturalness. However, as presented in Section 8.3.4, it is found that some architecture rules are rated poorly in all aspects. The rating of the aspects depends on several factors. By investigating the notes taken during discussions, the answers to the open-ended questions in the survey, and the interview transcriptions using qualitative analysis these factors are classified into the following categories:

Familiarity with architecture concepts and relations: The names of architecture concepts and relations greatly impact the understandability of an architecture rule. Participants may not be familiar with the concepts and relations used in the formalization, since examples from literature, open source systems, and websites are used that relate to types of software systems participants are not familiar with.

***ArchCNL* grammar and semantics:** Some keywords used in *ArchCNL* cannot appropriately represent the rule intention. Therefore, the rule formalization appears unnatural and is not understandable. For example, the rules `No Controller can access a DataAccessObject.` and `Nothing can access a Controller.` are rated as rather unnatural (mean = 4.5) and mostly not understandable (mean = 4.0). Based on the discussions, it can be observed that the keywords `no` and `can` are too weak in order to represent negation. Participants suggested that keywords like `must not` or `cannot` are more suitable to express negations.

Another example is the keyword `Every`, a specifier that is used for introducing the architecture concept for which an architecture rule is defined. This keyword indicates that there are several instances of an architecture concept. However, this assumption does not always apply for a rule. For example, the concept *application* is used to describe the software system that is developed. However, there is only one application for which the rules are described.

Chosen terms for concepts and relations: Although participants might be familiar with the concepts and relations used in the formalization, the terms might be chosen inappropriately for the formalization in *ArchCNL*. Some participants mentioned that passive voice for architecture relations is not understandable and should be avoided.

Structural complexity: As described in the previous section, some rules have a high structural complexity (Group 4). Although rules have rather short formulations in natural language, they result in a long and complex *ArchCNL* formalization. For example, due to the inherent complexity of the rules, variables, relative clauses, and coordinators applied in the formalization. Additionally, concepts and relations that have a commonly known meaning in natural language need to be explicitly specified in *ArchCNL*. This implies additional concepts and relations leading to a structural complex *ArchCNL* formalization. It seems that the participants have expected that the corresponding *ArchCNL* formalization is equally short. Consequently, the rule was harder to understand and less natural in *ArchCNL* than the given natural language description.

8.3.7. Threats to Validity

In this part, the threats to validity [WRH+12] of the study are discussed.

Conclusion validity: To ensure the reliability of measures, workshop participants had the chance to discuss the *ArchCNL*-based formalization during the workshop in order to resolve any uncertainties of the presented rules.

Internal validity: To address the maturation threat, the workshop format was first tested with two experts. Based on their feedback the workshop was refined, e.g., the number of presented rules was reduced to keep the workshop on the previously defined timeframe of 2 1/2 hours. To address the selection threat, workshop participants, although from the same company, come from different projects and different domains, and thus have different industrial background.

Construct validity: To ensure validity of the constructs, the research questions are clearly defined and methods for collecting data to address the research questions are systematically selected. Each aspect of applicability is covered by a question in the questionnaire, where the participants rate how they perceive each aspect, namely understandability, usability, and naturalness for each architecture rule presented.

The selection of architecture rules presented in the workshop could be a possible threat to construct validity. Architecture rules from different sources, such as literature, websites, or exemplary systems have been used. The evaluation of applicability could turn out differently if architecture rules from software projects in which the participants are involved in would have been formalized. As not all software projects the participants work in maintain architecture documents, examples from other sources have been used.

External validity: In this study, it is not claimed to achieve external validity of the findings due to several factors. The number of workshop participants (12 practitioners) is too small to generalize the results. Consequently, more empirical studies are needed to generalize results in the future. This means that the approach has to be used in more industrial projects in order to get better insights which kinds of architecture rules are used in industrial practice, which features are required by *ArchCNL*, and how the language needs to be designed to better reflect the actual needs of its users. Nevertheless, the preliminary findings already provide valuable input for future research and have helped to identify open research challenges.

Participants are all working in the same country and work in a company that has a strong focus in research. There might be potential differences in the attitude towards the approach compared to other participants working in another country and in another organization. Participants working in the investigated company could be more open to the proposed approach – since it is a research centre – than practitioners that do not work in a research context.

8.4. Architecture Violation Detection Quality

In this section, the architecture violation detection quality of *ArchCNLCheck* is evaluated using two case studies, namely TEAMMATES [tea] and JabRef [jab]. This section aims for answering the following research question (RQ):

RQ: To which degree is the approach able to detect relevant architecture violations?

In the following section, the study design (Section 8.4.1) is described. Selection criteria for the investigated case studies and the measures for evaluating the violation detection quality are defined. The results of the investigated case studies are presented in separate sections (Section 8.4.2 and Section 8.4.3).

8.4.1. Evaluation Steps and Measures

The goal of this evaluation is to validate the architecture violation detection quality of the proposed approach. Architecture detection quality is defined as follows:

Definition 8.4.1: Architecture Detection Quality

Architecture detection quality is the degree to which a conformance checking approach is able to detect relevant architecture violations.

In order to evaluate the architecture detection quality of *ArchCNLCheck*, the following steps are performed:

1) Selecting the Case Studies: The case studies have been chosen based on the following criteria:

- The source code of the software system is available, since the approach requires the source code files to create the ontology-based code model.
- The system is developed in Java, since the approach currently provides support for parsing Java-based implementations.
- Architecture rules of the software systems are documented, e.g., as part of a software architecture documentation. Architecture rules can be defined formally, e.g., using a specification language of a conformance checking tool, or informally, e.g., in natural language. This documentation is necessary to design the architecture ontology, i.e., the architecture concepts for which architecture rules are defined and the relations that connect the architecture concepts.
- A reference data set with detected violations is available for the software system. This data set is considered as a so-called *ground truth* of confirmed architecture violations that are known to exist in the software system (in a specific version). This ground truth is used for comparing the results detected with *ArchCNLCheck* in order to evaluate the architecture violation detection quality.
- The investigated software system is not trivial, i.e., the size of the software system is "sufficiently large" and the software system is developed over a longer period of time. "Sufficiently large" means that the size of the software is too large for manual inspection, i.e., tool-supported analysis is necessary to detect architecture violations. In the context of this evaluation, the size of a software system is measured in terms of Lines of Code (LoC), number of commits, number of contributors involved in the development of the software system, the age of the software system in years, and the number of defined architecture rules. These quantitative measures can be justified as follows. LoC is a commonly used measure and indicator for the complexity of a software system. For selecting an appropriate case study, projects with a size of ≥ 100 kLoC are targeted. It is important to note that the LoC metric is not always an appropriate indicator for software complexity and the size of a software system [Kem93]. That is why, the other measures are additionally used to characterize the size of a project. The higher the

Table 8.5.: *Overview on the investigated case studies.*

Case study	kLoC	# commits	# contributors	Age (Years)	Version	#Rules	#Rules Violated
TEAMMATES	$\sim$ 140	$\sim$ 17k[3]	403	9	5.110	34	13
JabRef	$\sim$ 157	$\sim$ 13k[4]	220	16	3.7	17	9

number of developers involved in the implementation, the higher could be the risk that violations are introduced. At minimum, at least 10 developers should be involved. The number of commits roughly corresponds to the changes made to the software system. The more changes have been made to the system, the higher is the risk of introducing architecture violations. For the evaluation, software systems with at least 5000 commits are selected. Lastly, the number of architecture rules defined in the projects are used as another indicator for the size of the project. The more architecture rules are defined, the more tedious a manual validation of the rules becomes. Therefore, the automatic validation of architecture rules becomes obligatory. For the evaluation, software systems with at least 10 rules are selected. The age of the software system is measured in years. At minimum, the software should be developed for 5 years. This criterion ensures that some architecture evaluation has occurred.

Based on these criteria, two open-source software systems have been selected for evaluating the architecture detection quality. Table 8.5 depicts an overview on relevant data of the case studies.

TEAMMATES TEAMMATES is an open source project and is a web-based feedback management tool for education. It is chosen, since it provides an up-to-date comprehensive documentation of the architecture design and is a rather complex system (140 kLOC). In particular, the architecture rules are documented. The documentation provides all the necessary information needed to build the architecture concept language and to formalize the architecture rules of TEAMMATES. TEAMMATES is developed in Java. TEAMMATES has been developed since 2010.

JabRef JabRef is an open source bibliography reference manager. The native file format used by JabRef is BibTeX, the standard LaTeX bibliography format. Architecture rules are documented as a text-based package diagram depicting the permitted dependencies between the packages JabRef consists of. This model is used to derive the architecture rules. JabRef is developed with Java. JabRef has been developed since 2003.

2) Architecture Rule Formalization: Both case studies selected for the evaluation maintain architecture documentations containing architecture rules. In TEAMMATES, architecture rules are defined in an XML-based file that is used for conformance checking. The TEAMMATES developers use the tool Macker [mac] for validating the architecture rules. In JabRef, the intended architecture is described as a text-based package diagram. It describes the packages

[3]Last access: August 2019
[4]Last access: August 2019

structuring the source code and the allowed dependencies between the packages. Those resources are used to design the architecture ontology. First, architecture concepts are identified for which the rules are defined. Then, the architecture relations connecting the concepts are identified. For the concepts and relations, appropriate names are chosen to use them in the rule formalization. Ideally, terms used in the documentation are reused in the ontology-based rule formalization. Having identified the architecture concepts and relations, the architecture rules are formalized using *ArchCNL*. The architecture rules defined in both case studies belong to the *static dependency rule* category (see Section 8.2.4). These rules prescribe the prohibited (is not allowed to use), permitted (is allowed to use), and the obligatory (must use) static dependencies between architectural elements, e.g., modules or packages.

3) Formalizing the Architecture-to-Code-Mapping: In a subsequent step, the identified concepts and relations need to be mapped to the code. While the mapping of the architecture concepts is specific for each case study, the architecture relation *use* and its mapping to the code can be defined generally for both case studies. Table 8.6 depicts the architecture-to-code-mapping of the *use* relation formalized as SWRL rules. The *use* relation is derived from a code model based on the FAMIX ontology. The architecture-level *use* relation is derived as follows from the code model:

Return Type of Method: If an individual x is a TYPE (i.e., a class, an annotation, or an enum) that defines a method m which returns y, then x uses a y.

Attribute Declaration: If an individual x is a TYPE that defines an attribute a and this attribute has the declared type y, then x uses y.

Invocation: If an individual i is an invocation that has a sender x that is a TYPE (i.e., the type of the object that invokes a specific method) and that has a receiver r (i.e., the object on which the method is invoked) which has the type y, then x uses y.

Import Declaration: If an individual x is a TYPE that imports y, then x uses y.

Local Variable Declaration: If an individual x is a TYPE that defines a method m in which a variable v with the declared type y is defined, then x uses y.

Parameter Declaration: If an individual x is a TYPE that defines a method m defining a parameter p which has a declared type y, then x uses y.

Applying a rule-based reasoner on the ontology-based code model infers the *use* relation between the relevant individuals (as described in the mapping before).

Architecture concepts are inferred based on regular expressions on package and class names. Table 8.7 depicts the corresponding templates for the mapping that is applied in the case studies to derive the architecture concepts. Two conventions are used in the case studies for the concept mapping:

Type Name: If an individual x is a TYPE which has a name matching a specific string pattern, then x is inferred to be an individual of the architecture concept C.

Package Containment: If an individual x is a TYPE which is contained in n which is a NAMESPACE where n has a name matching a specific string pattern, then x is inferred to be an individual of the architecture concept C.

Table 8.6.: *Mapping of the architecture-level use relationship to code-level relationships. Atoms in the rules refer to concept and relation names of the FAMIX ontology defined in Chapter 7.*

Code-level relationship type	Mapping Rule
Method Return Type	$Type(x), Type(y), Method(m),$ $definesMethod(x, m), hasDeclaredType(m, y)$ $\rightarrow use(x, y)$
Attribute Declaration	$Type(x), Type(y), Attribute(a),$ $definesAttribute(x, a), hasDeclaredType(a, y)$ $\rightarrow use(x, y)$
Invocation	$Type(x), Type(y), Invocation(i),$ $hasSender(i, x), hasReceiver(i, r), hasDeclaredType(r, y)$ $\rightarrow use(x, y)$
Import Declaration	$Type(x), Type(y), imports(x, y) \rightarrow use(x, y)$
Local Variable Declaration	$Type(x), Type(y), Method(m),$ $definesMethod(x, m), Variable(v),$ $definesVariable(m, v), hasDeclaredType(v, y)$ $\rightarrow use(x, y)$
Parameter Declaration	$Type(x), Type(y), Method(m),$ $definesMethod(x, m), Parameter(p),$ $definesParameter(m, p), hasDeclaredType(p, y)$ $\rightarrow use(x, y)$

4) Calculating Architecture Violations: For detecting architecture violations, the toolchain (see Section 7.6) that implements the process of *ArchCNLCheck* is applied to the two software systems. The source code files, the mapping rules (Table 8.7 and Table 8.6), and the architecture rules are provided as input. The architecture rules and the mapping rules are provided in an ascii-doc-based [asc] text file that is automatically transformed by the toolchain into the architecture and mapping ontology. The output is a text file containing the architecture violations.

5) Evaluating the Architecture Detection Quality: In order to evaluate the quality of the violation detection of the approach, the detection results of the tool HUSACCT [PKvdWB14] that was applied during the SAEroCon Workshop[5] are used for the systems TEAMMATES and JabRef. The results are used as a ground truth in order to compare them with the architecture violations found with *ArchCNLCheck*. The results of the architecture analysis were documented during the workshop and are available at its repository[6]. Each single violation in the ground truth has been verified and confirmed by the responsible software architect of the respective software system. Based on this, the results obtained with *ArchCNLCheck* can be compared with the ground truth as described in the following.

The ground truth contains the following violation types:

[5]https://saerocon.wordpress.com/

[6]https://github.com/sebastianherold/SAEroConRepo/wiki

Table 8.7.: *Mapping of architecture concepts to code-level concepts. Atoms in the rules refer to concept and relation names of the FAMIX ontology defined in Chapter 7.*

Mapping Convention	Mapping Rule
Type Name	$Type(x), hasName(x, name), regex(name, PATTERN) \rightarrow C(x)$
Package Containment	$Type(x), Namespace(n), hasName(n, name), regex(name, PATTERN), ?namespaceContains(n, x) \rightarrow C(x, y)$

- divergences, i.e., violating types for each rule that establish a forbidden *use* relationship to another type, and
- absences, i.e., types that miss to establish a *use* relation to another type.

For each rule that has been violated, the violating types detected with the ontology-based approach are compared manually with the set of violating types in the ground truth. The types are compared with each other by their full qualified name. The violations detected with the *ArchCNLCheck* are classified as *true positive* (TP), i.e., true violations according to the ground truth, *false positive* (FP), i.e., violations detected with the ontology-based approach but not contained in the ground truth, and *false negative* (FN), i.e., violations that are contained in the ground truth, but are not detected by *ArchCNLCheck*. Based on the values for TP, FP, and FN, the measures *precision* and *recall* are calculated. These measures are used as surrogates for the architecture detection quality of the approach. Precision P is calculated as

$$P = \frac{TP}{TP + FP} \tag{8.1}$$

whereas recall R is calculated as

$$R = \frac{TP}{TP + FN} \tag{8.2}$$

Precision can be understood as the fraction of relevant architecture violations among the retrieved architecture violations. Recall is the fraction of relevant architecture violations that have been retrieved among the total amount of relevant architecture violations.

Additionally, a prioritization of the discovered violations is available in the ground truth. This prioritization is used in order to evaluate whether crucial violations are detected.

8.4.2. TEAMMATES

In this section, the conformance checking results of TEAMMATES are presented. The TEAMMATES developers specify architecture rules in an XML file that is used as input for the tool Macker that validates these rules. In total, 34 rules are formalized in this file. In the HUSACCT data set, 41 rules are formalized, i.e., more rules than specified by TEAMMATES. This is due to the fact that some rules of TEAMMATES are mapped to several module dependency rules in HUSACCT. Examples of such formalizations are depicted in Table 8.8.

By comparing the rules defined in the XML file and the rules defined in the HUSACCT data set, it can be found that not all rules of TEAMMATES are specified by HUSACCT. That is why, both sources are used for the rule formalization to cover all relevant rules for conformance checking.

In the following, it is shown how an architecture ontology for the TEAMMATES architecture is designed containing architecture concepts necessary to formalize the rules. Additionally, a concrete mapping example is shown for concepts and relations according to the patterns described in Table 8.6 and 8.7. The HUSACCT formalization and the formalization in *ArchCNL* is shown. Examples of architecture rules not covered in the HUSACCT data set and their corresponding formalization in *ArchCNL* are presented. Based on the results obtained in this case study, the architecture violation detection quality of the approach is assessed.

Architecture Ontology and Rules of TEAMMATES: Figure 8.6 shows an excerpt of the architecture of TEAMMATES. The TEAMMATES developers documented all the important architecture rules that need to be followed by the implementation. The documentation is used in order to formalize the rules with the *ArchCNL* and to check them against the source code. In the following, it is demonstrated how the architecture rules are formalized using the approach by 1) defining an ontology of the main architecture concepts of TEAMMATES (see Figure 8.7), 2) formalizing the architecture rules based on the concepts (see Table 8.8 and Table 8.9), and 3) formalizing the architecture-to-code-mapping between the architecture and the code concepts using SWRL rules so that the architecture conformance checking can be performed using reasoning services (see Figure 8.8).

Figure 8.7 shows the project-specific architecture concepts of TEAMMATES with which its architecture is described and for which architecture rules are defined. Four main categories of architecture concepts have been identified. The ontology defines the concept *TeammatesThing* as a so-called “root concept” in order to designate the architecture concepts belonging to the TEAMMATES architecture. Every concept that should become a part of the TEAMMATES architecture concept language must be a sub-concept of the root concept. The following TEAMMATES concepts have been identified

- *TeammatesComponent* is a concept representing a high-level structure element of the architecture of TEAMMATES. *UILayer*, *StorageLayer*, *CommonComponent*, and *LogicLayer* are specific types of *TeammatesComponent*s. The concept *ExternalComponent* is used to summarize types that originate from external libraries.

- *TeammatesType* is considered the smallest unit the TEAMMATES architecture can be defined with.

- *TeammatesPackage* is a *TeammatesThing* that resides within a *TeammatesComponent* and structures the component.

- *TeammatesAPI* models the public interface of a *TeammatesComponent*.

After having identified the main architecture concepts, the architecture rules are defined based on those concepts. Selected architecture rules and their corresponding formalization in *ArchCNL* are listed in Table 8.8 and Table 8.9. For the selected subset in Table 8.8 and Table 8.9 the *Negation*, the *If-Then*, the *Can-Only*, and the *Only-Can* rule types are used.

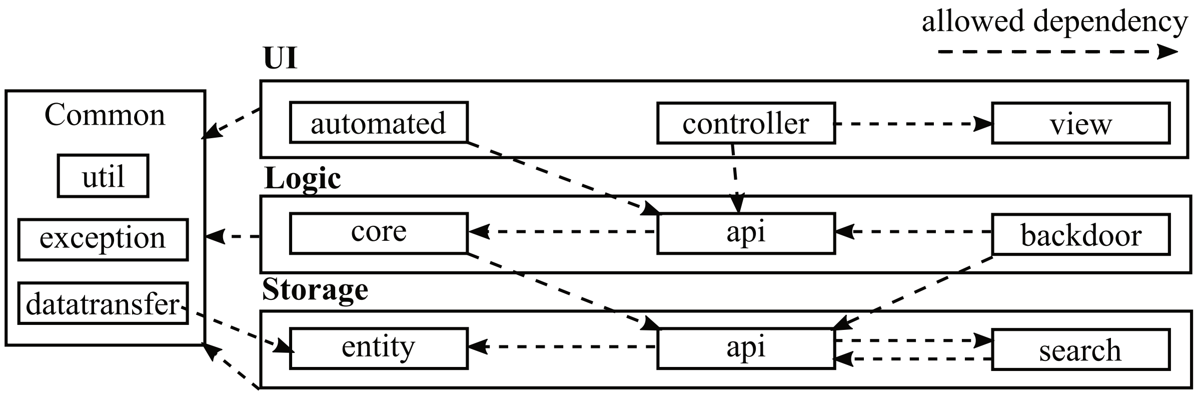


Figure 8.6.: *Excerpt from the architecture design of TEAMMATES.*

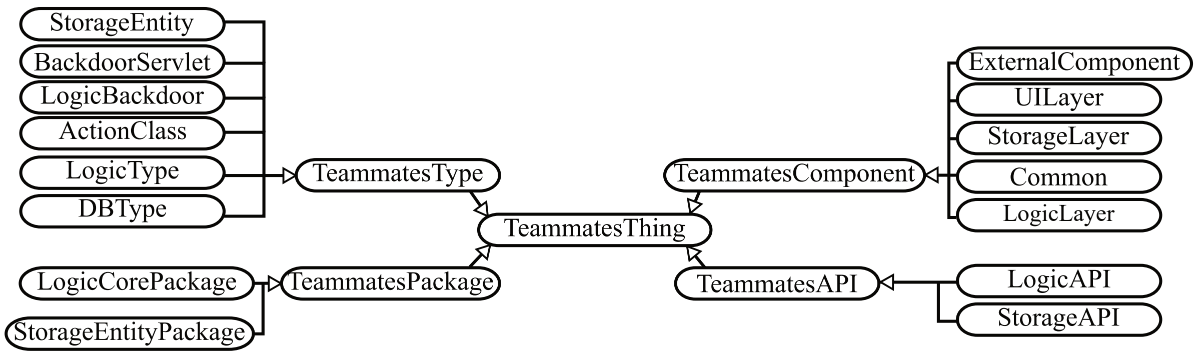


Figure 8.7.: *Architecture ontology of TEAMMATES.*

Architecture-to-Code-Mapping in TEAMMATES: The developers of TEAMMATES use naming conventions in order to designate architecture concepts in the source code. As can be observed in Table 8.9, an architecture relation *manage* for rule **R7** – in addition to the *use* relation – is introduced. This relation emphasizes that each *LogicType* has a corresponding *DBType*. The developers apply the convention that the corresponding *DBType* needs to contain the same prefix as the *LogicType*. For example, the class implementing the *LogicType* "CourseLogic" has the corresponding *DBType* "CourseDb" (they both contain the prefix "Course"). This convention is exploited in order to derive the architectural-level *manage* relationship between *LogicType* and *DBType* individuals. In Figure 8.8 the mapping rules for the concept *LogicType* and the architecture-level *manage* relationship are shown that are mapped by such a naming convention. Intuitively, the mapping rules can be understood as follows:

- **(A)**, *LogicType* mapping: If something is a *FamixClass* and has a name containing "Logic" as its suffix, then this class is identified as a *LogicType*. The concept *DBType* is mapped accordingly by identifying classes that have a name ending with "Db".
- **(B)**, *manage* mapping: The *manage* relationship is derived by matching regular expression patterns based on the names of classes representing instances of the *LogicType* and *DBType* concepts. The function *regex* matches the lexical form of a literal against a regular expression pattern given by another literal. By successfully matching a regular expression pattern, the function binds the matched literal to capture groups, if any are defined. In this example, the *regex* function defines two capture groups. For example, for the name of the *LogicType*, the variable $m1$ representing the first capture group is bound

LogicType
manage
DBType

(A) LogicType Mapping

$FamixClass(?class), hasName(?class, ?name), regex(?name, '\backslash\backslash w * Logic') \rightarrow LogicType(?class)$

(B) manage Mapping

$LogicType(?class), hasName(?class, ?logicName), regex(?logicName, '(\backslash\backslash w*)(Logic)', ?m1, ?m2),$
$DBType(?classDB), hasName(?classDB, ?dbName), regex(?dbName, '(\backslash\backslash w*)(Db)', ?m3, ?m4),$
$equal(?m1, ?m3) \rightarrow manage(?class, ?classDB)$

Figure 8.8.: *Mapping the architecture concepts LogicType and the architecture-level manage relationship to object-oriented code concepts.*

to the prefix of the name. The second capture group is bound to the literal “Logic” which the suffix of the name of a *LogicType*. The matching for the name of a *DBType* works correspondingly. The variable $m3$ is bound to the prefix of the name of the *DBType*. The function *equal* compares the two literals bound to $m1$ and $m3$. If they are equal, then a *manage* relation between the *LogicType* and the *DBType* can be derived.

Architecture Conformance Checking Results: The architecture rules are checked against version 5.110 of TEAMMATES. This is the same version that was also used to create the HUSACCT results that constitute the ground truth for the evaluation. The ground truth does not formalize all architecture rules defined by the development team of TEAMMATES. Furthermore, additional violations have been found that are not contained in the ground truth. These results will be explained in a separate section.

The ground truth contains violations for 11 out of 41 architecture rules. For exactly those 11 rules, violations have been found with *ArchCNLCheck*. Some of the found violations, especially the dependency between the classes `FeedbackResponsesLogic` and `FeedbackResponse` (implementing the *StorageEntity* concept) have been described as *“severe and should be fixed”* by the architect of TEAMMATES during the workshop discussion. This dependency violates the rule `No LogicLayer can use a StorageEntity`. This is a severe violation, since it violates the strict layer pattern [BMR^{+}96]. The violated rules and the amount of violations found (with *ArchCNLCheck* and with the HUSACCT tool) are depicted in Table 8.10. No false positives have been found for those rules, i.e., architecture violations that are not contained in the ground truth. However, it can be observed that not all violating classes could be revealed with *ArchCNLCheck*. This is due to missing information in the code model and in the architecture-to-code-mapping, respectively. HUSACCT is designed to provide a comprehensive detection of dependencies between code elements [PKvdWB17] and is therefore able to provide detailed information on detected violations. The parser responsible to transform the source code to the code model (FAMIX) needs to be extended correspondingly to the dependency types covered by HUSACCT. However, a significant trade-off between the amount of information covered by a code model and the time complexity of the conformance checking process must be considered. The more information is extracted from the code, the more triples in the fact base need to be processed. As a consequence, reasoning used for detecting architecture violations may become more time intensive.

Table 8.8.: *An excerpt of architecture rules defined by the TEAMMATES development team given in natural language, their formalization in HUSACCT (left column) and using ArchCNL (right column). The arrow $\nrightarrow$ refers to the* "is not allowed to use" *predicate in HUSACCT, whereas the arrow $\rightarrow$ refers to the* "is allowed to use" *in HUSACCT. The arrows are used for the purpose of abbreviation. UI.*, Logic, Common.*, Client, TestDriver, Storage.* are modules defined in HUSACCT.*

	HUSACCT	**ArchCNL**
	UI should not touch storage	
R1	UI is not allowed to skip call	`No UILayer can use a StorageLayer.`
	Logic should not touch UI	
R2	Logic is not allowed to back call	`No LogicLayer can use a UILayer.`
	Common should not have dependencies to any packages except storage::entity	
R3	Common.datatransfer $\nrightarrow$ Storage Common.datatransfer $\rightarrow$ Storage.entity Common.exception $\nrightarrow$ Storage Common.util $\nrightarrow$ Storage Common $\nrightarrow$ Client Common $\nrightarrow$ Logic Common $\nrightarrow$ TestDriver Common $\nrightarrow$ UI	`Every CommonComponent can only use a CommonComponent or use a StorageEntityPackage.`
R4	*Only *Action classes can touch Logic API*	
R4	Client $\nrightarrow$ Logic Common $\nrightarrow$ Logic TestDriver $\nrightarrow$ Logic UI.view $\nrightarrow$ Logic	`Only an ActionClass can use a LogicAPI.`
	Controllers should be self-contained	
R5	Client $\nrightarrow$ UI Common $\nrightarrow$ UI TestDriver $\nrightarrow$ UI UI.view $\nrightarrow$ UI.controller	`Only a Controller can use a Controller.`
	storage::entity should not depend on anything within the storage component	
R6	Storage.entity $\nrightarrow$ Storage.api Storage.entity $\nrightarrow$ Storage.search	`No StorageEntityPackage can use a StorageAPI or use a StorageSearchPackage.`

ArchCNLCheck is currently restricted to those dependency types shown in Table 8.6. For example, variable accesses are not yet covered in the code model and in the mapping.

Based on the values for TP, FN, and FP, precision and recall are calculated for those rules. The resulting precision is 1.0, i.e., the best value for precision that can be achieved. This means that the violations that have been detected are all relevant. This is due to the fact that no false positives have been detected. The resulting recall is 0.77. This means that 77% of the relevant violations have been found. As can be seen in Table 8.10 there are some false negatives, i.e., some relevant architecture violations have not been detected. However, the recall is considered as appropriate. The recall could be further improved by extending the parser – e.g. by including variable accesses – to contain more detailed information on dependencies between code elements, so that more violations can be detected.

As described before, not all architecture rules as defined by the TEAMMATES developers are contained in the ground truth. This is due to the fact that not all rules can be expressed with HUSACCT. Nevertheless, all 34 rules defined in TEAMMATES have been formalized with *ArchCNL* and validated with *ArchCNLCheck*. For some of those, additional rules architecture violations have been found as described in the following. Table 8.11 summarizes the results.

For rule **R4** (see Table 8.8) 23 violations have been found whereas five of them were not detected by HUSACCT. Four of them are attributed to so-called servlet classes that touch the *LogicAPI*, although they are not classified to be individuals of the *ActionClass* concept. The remaining one stems from the `BackDoorLogic` class that also touches *LogicAPI* without being an *ActionClass*. The HUSACCT formalism did not find those violations since neither of the classes have been included in the formalization of this rule. **R7** (see Table 8.9) is violated ten times by the class `BackDoorLogic`, since it uses classes that implement the *DBType* concept. However, class `BackDoorLogic` does not manage those *DBType*s (`BackDoorDb` does not exist). Those are real violations found by the approach that cannot be found with the HUSACCT formalism, since it does not support the conditional rule type (see Table 6.1, Chapter 6). Three more architecture violations have been found for rule **R11** (see Table 8.9), another rule that is not included in the ground truth. Although not included in the ground truth, these “false positives” should be considered as real violations, i.e., true positives.

8.4.3. JabRef

In this section, the conformance checking results of JabRef are presented. Architecture rules are documented in a Wiki[7] of the respective GitHub Repository the JabRef source code is hosted on. The architecture rules are documented as a text-based package diagram maintaining the allowed *use* dependencies between the packages. In total, JabRef defines 17 architecture rules. In contrast to TEAMMATES, these rules are completely covered in the ground truth. In the following, the resulting architecture ontology representing the architecture concept language of JabRef are presented. The same mapping conventions for architecture-to-code-mapping as in TEAMMATES are applied. According to the assessment in TEAMMATES, the architecture violation detection quality of *ArchCNLCheck* is assessed based on precision and recall.

[7]https://github.com/JabRef/jabref/wiki/High-Level-Documentation

Table 8.9.: *Architecture rules defined by the TEAMMATES development team that either have* ***not*** *been formalized with HUSACCT or for which additional violations have been found that are not contained in the ground truth. The corresponding formalization in ArchCNL is given in* `typewriter font` *for each rule description.*

R7	*Each logic can only access its corresponding DB (e.g. AccountsLogic and AccountsDb)* `If a LogicType uses a DBType, then it must manage this DBType.`
R8	*Test cases should not depend on each other* `No TestCase can use a TestCase.`
R9	*Only UI tests can access page object classes.* `Only a UITest can use a PageObjectClass.`
R10	*Google Cloud Storage API can only be accessed via Google Cloud Storage Helper.* `Only a GoogleCloudStorageHelper can use a GoogleCloudStorageAPI.`
R11	*Servlet API can only be accessed via Servlet classes, GaeSimulation, and selected utility classes.* `Only a ServletClient can use a ServletAPI.`

Architecture Ontology and Rules of JabRef: Figure 8.9 depicts the architecture model of JabRef illustrated as a package diagram. The JabRef developers define permitted static dependencies between these packages that need to be respected in the implementation. In the tested version (3.7) the developers have not applied any automatic analyses to verify these dependencies. The architecture of JabRef follows the relaxed layer pattern [BMR$^+$96]. The implemented layer pattern is not strict, since upper layers are allowed to skip adjacent layers. The packages GUI, CLI, Logic, and Model are layers, whereas the packages Globals and Preferences are considered subsystems. The libraries Swing and SQL are external systems. Each package is considered as a candidate for representing an architecture concept of the ontology. Consequently, the following architecture concepts are defined as part of the architecture ontology:

- *Layer*: An architecture concept representing a layer, i.e., a unit of logical separation in the software architecture where each layer has a specific and uniquely assigned responsibility in the software architecture. Layers have a hierarchical level and need to follow rules on the use of layers on another hierarchical level.
- *Subsystem*: An architecture concept representing a subsystem, i.e., a unit in the software architecture with clearly assigned responsibility.
- *GUILayer* is a layer responsible for graphical user interface functionality.
- *CLILayer* is a layer responsible for the command line interface of JabRef.

Table 8.10.: *Architecture rules of TEAMMATES that have been formalized and validated by HUSACCT, the number of violations detected with ArchCNLCheck, and the number of found violations as contained in the ground truth (GT).*

Architecture Rule (in ArchCNL)	**# Violations Found**		**# Violations in GT**
	TP	FN	
No Client can use a LogicLayer.	16	0	16
No Client can use a StorageLayer.	25	1	26
No Client can use a TestDriver.	26	11	37
No CommonClass can use a LogicLayer.	7	0	7
No CommonClass can use a GUILayer.	14	0	14
No LogicLayer can use a StorageEntityType.	1	0	1
Only StorageAPI can use a ObjectifyAPI.	53	0	53
No TestDriver can use a LogicLayer.	122	20	142
No TestDriver can use a StorageLayer.	63	2	65
No TestDriver can use a GUILayer.	295	231	526
No GUILayer can use a LogicBackdoor.	1	0	1
Total	**690**	**198**	**888**
Precision		**1.0**	
Recall		**0.77**	

- *LogicLayer* is a layer responsible for manipulating instances of the *ModelLayer* concept.
- *ModelLayer* is a layer containing the most important data structures that model the application domain of JabRef.
- *GlobalSubsystem* is considered a subsystem containing functionality which is required by several components of JabRef, i.e., layers and subsystems, of the software system.
- *PreferenceSubsystem* is considered a subsystem representing and storing all information that is customizable by a user.
- *SwingLibrary* is an external component representing classes of the Java Swing library[8] for implementing graphical user interfaces.
- *JavaSQLAndOracleLibrary* is an external component representing classes of an SQL library for implementing database accesses.

Table 8.12 depicts the corresponding architecture rules for those architecture concepts. Similar to the TEAMMATES case study, architecture rules are formalized in terms of static dependency rules. This means that architecture concepts are connected by the architecture-level *use* relationship. It is important to note that not all architecture rules are shown here, but only those rules for which violations exist.

[8] `https://docs.oracle.com/javase/8/docs/api/index.html?javax/swing/package-summary.html`

Table 8.11.: *Results of the conformance check for the rules **R4**, **R7**, and **R11**. For these rules, false positives (FP) have been found. However, these violations should be considered as real violations (TP), since these rule are either not formalized in HUSACCT or there are classes that have not been included in the conformance check with HUSACCT.*

Rule	ArchCNL	TP	FP
R4	`Only an ActionClass can use a LogicAPI.`	18	(5)
R7	`If a LogicType uses a DBType, then it must manage this DBType.`	0	(10)
R11	`Only a ServletClass can use a ServletAPI.`	0	(3)

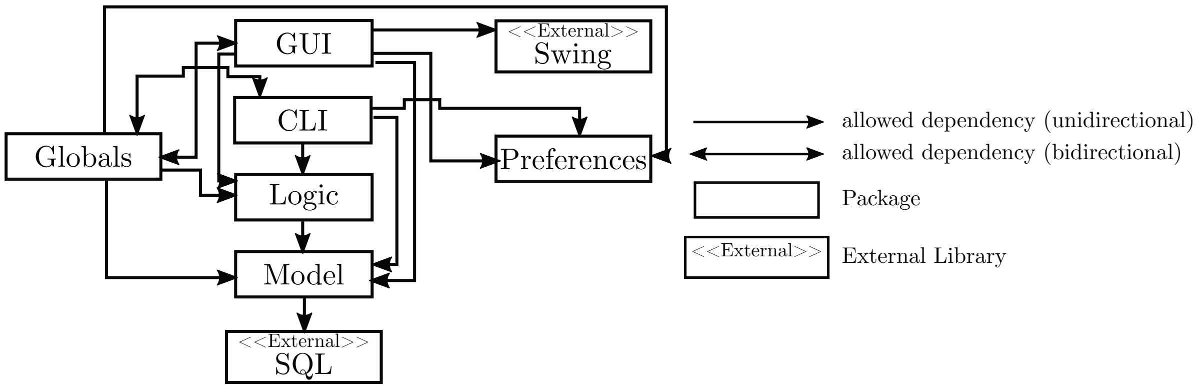


Figure 8.9.: *Excerpt of the software architecture of JabRef captured as the package structure and the allowed dependencies between the packages.*

Architecture-to-Code-Mapping: As described by the JabRef developers, layers and subsystems are represented by the respective packages and classes that reside in the package. For example, the GUI layer is implemented by the package named `gui`. Consequently, the naming and package conventions as described in Table 8.7 can be reused in this case study. Correspondingly, the mapping rules for the *use* relationship presented in Table 8.6 are applied.

Architecture Conformance Checking Results: The ground truth of JabRef contains violations for 9 out of 17 architecture rules. For those 9 rules, violations also have been found with *ArchCNLCheck*. The violated rules and the amount of violations found (with *ArchCNLCheck* and with the HUSACCT tool) are depicted in Table 8.12. For those rules, no false positives have been found. Due to missing information in the code model, e.g., accesses to variables, not all architecture violations contained in the ground truth could be revealed. This can potentially be resolved by a more extensive formalization as already discussed in the results paragraph of Section 8.4.2.

As for the TEAMMATES case study, the precision of the approach is 1.0, since no false positives have been detected. Considering recall, 91% of the relevant violations have been found. The recall value is close to the best value (= 1.0). This means that the approach is able to detect nearly all relevant violations out of all relevant violations in the context of this case study.

Table 8.12.: *Architecture rules of JabRef, the number of violations detected with ArchCNLCheck, and the number of violations contained in the ground truth (GT). Only rules are listed for which violations have been found.*

Architecture Rule (in ArchCNL)	**# Violations Found**		**# Violations in GT**
	TP	FN	
No GUILayer can use a CommandLineInterfaceLayer.	3	1	4
Only a GUILayer can use a SwingLibrary.	4	0	4
Only a LogicLayer can use a JavaSQLAndOracleLibrary.	7	0	7
No LogicLayer can use a GUILayer.	3	0	5
No LogicLayer can use a PreferenceSubsystem.	8	0	8
No ModelLayer can use a LogicLayer or a GUILayer.	3	0	3
No ModelLayer can use a PreferenceSubsystem.	1	0	1
No PreferenceSubsystem can use a GUILayer.	8	1	9
No PreferenceSubsystem can use a GlobalSubsystem.	2	0	2
Total	**39**	**4**	**43**
Precision		**1.0**	
Recall		**0.91**	

8.4.4. Discussion on the Architecture Detection Quality

The goal of this evaluation section is to determine the architecture violation detection quality of the *ArchCNLCheck*. The evaluation results, i.e., precision and recall, show that the approach is adequate for checking the conformance of the implemented architecture against architecture rules. All in all, based on the evaluation results, the research question

> **RQ: To which degree is the approach able to detect crucial architecture violations?**

is answered as follows:

Evaluation: Answer to RQ

The approach is able to detect a great amount of relevant architecture violations in both case studies. For TEAMMATES and JabRef the **precision** of the approach is **1.0**. This means that the approach has detected violations that are all relevant, i. e., there are no false positives. Concerning **recall**, the approach is able to detect **77%** of the relevant violations in the TEAMMATES case study, whereas for the JabRef case study the approach is able to even detect **91%** of the relevant violations.

Concerning the rule formalization, *ArchCNL* has shown to be more precise than the reference tool. In the TEAMMATES case study, it was shown that in contrast to HUSACCT, the approach is able to formalize all architecture rules that are defined in the documentation of TEAMMATES. As a result, architecture violations have been found that have not been detected with HUSACCT. This shows how a restrictive meta model of a specification language can lead to important violations remaining undetected.

Nevertheless, the architecture detection quality could be further improved by extending the parser functionality for more detailed information in the code model. Since some code-level information is not covered in the code model, e.g., variable accesses, some violations could not be detected with the approach.

8.4.5. Limitations of the Study

Promising results for precision and recall have been measured implying a suitable level of violation detection quality. However, the general validity of the measures is not assured due to the limited number of observations. In order to assure the general validity and to ensure the statistical relevance, more case studies need to be conducted.

Architecture rules formalized and validated in this case study can be mainly characterized as dependency rules. In order to validate whether the approach is able to find architecture violations from other categories – as those found in the study in Section 8.2 – more case studies need to be conducted that define a greater variety of architecture rules.

Lastly, the quality of the ground truth greatly determines the results of precision and recall of the conformance checking. For example, there is a risk that there are crucial violations that have not been detected with HUSACCT and are therefore not included in the ground truth. As described earlier, this has led to a number of unjustified false positives (FP, see Table 8.11) in the evaluation of TEAMMATES. To mitigate this risk of missing violations, the defined architecture rules and the detection results contained in the ground truth have been thoroughly discussed with the responsible software architect of the respective software system during the SAEroCon workshop.

8.5. Evaluation Summary and Discussion

The goal of the evaluation is to determine how the approach fulfills the thesis goals **G2** and **G3** described in Chapter 1. In this section, the evaluation results are summarized and the implications for the fulfillment of the thesis goals are discussed. The findings of the study from Chapter 3 are used as criteria in order to judge to which extent the proposed approach integrates to the enforcement process and how it supports enforcement activities. Finally, a critical discussion on the limitations of the approach is given.

8.5.1. Summary

The results in Section 8.2 have confirmed the flexibility of the approach by showing that a great amount of architecture rules found in the investigated projects can be formalized with *ArchCNL*. Compared to other approaches, *ArchCNL* can reflect the project-specific language more appropriately, since architecture concepts and relations can be flexibly defined. As a future improvement, *ArchCNL* could be further extended by new language elements so that architecture rules can be formalized that are currently not supported by *ArchCNL*.

In Section 8.3, it has been shown that *ArchCNL* is perceived as an understandable approach for architecture rule formalization. Based on the results it can be concluded that the approach proves to be beneficial in terms of integrating formal architecture documentation – and combining them with informal documentation – by using natural-language frontends like *ArchCNL*. Thus, the evaluation has demonstrated that the approach allows for a formal, flexible, and understandable definition of the architecture concept language and therefore, the approach fulfills thesis goal **G2**. Nonetheless, more case studies and experiments are needed in order to make conclusions about the *actual* understandability of the approach.

The approach has an appropriate architecture detection quality. The results in Section 8.4 show that it is able to detect mostly all relevant architecture violations for both case studies. Thus, it is suitable to be used for validating whether the architecture concept language has been violated in the source code. As a consequence, the approach fulfills thesis goal **G3**. However, the case studies only define static dependency rules. In a next step, the approach should be tested on software systems that define rules from the other architecture rule categories as found in the study in Section 8.2.4. This allows to make conclusions about to which extent the approach is able to verify the conformance of the source code against other types of rules.

8.5.2. Support for Architecture Enforcement

The approach aims for supporting the architecture enforcement process. In Chapter 3 enforcement activity groups have been revealed that are part of architecture enforcement. In the following, the activity groups found in the study are listed and it is discussed to which extent the activities are supported by the approach.

Achieving Mutual Understanding of the Architecture: In the study presented in Chapter 3, the findings revealed that the awareness of software architecture and the mutual understanding about it are of crucial importance during architecture enforcement. In order to achieve mutual understanding, it is important to appropriately model the software architecture. By explicitly capturing and formally defining the language encompassing concepts and relations used to talk about the software architecture, this mutual understanding about architecture can be enhanced. Additionally, the language can be used for architecture documentation that is used by developers (activity “modeling software architecture for developers”). The approach greatly supports this enforcement activity by providing *ArchCNL* that allows software architects to formally define architecture concept languages. Due to its formality, sentences written in *ArchCNL* are unambiguous reducing the risks of misconceptions about the software architecture. Since *ArchCNL* is natural and understandable, *ArchCNL* facilitates the mutual understanding of defined architecture concepts, relations, and rules in the development team.

Providing Implementation Templates for Software Architecture: Providing implementation templates such as skeletons or by code generation is not the focus of the approach. However, the language and the rules defined with the approach can guide the architect to define appropriate templates for implementing the architecture concepts and relations. The architecture-to-code mapping indirectly corresponds to a template that defines how an architecture concept or relation have to be implemented in the source code. Since the architecture concept language is formalized, it can potentially be used for code generation.

Ensure Feasibility of the Architecture: The established architecture concept language serves as a means for discussion about chosen architecture concepts and defined architecture rules. Ideally, the language and the corresponding architecture rules are defined collaboratively by the software architects and the developers. This helps to directly gather feedback about the suitability of architecture rules and, if necessary, to adapt the concepts, relations, and corresponding rule definitions.

Awareness of Architecture in Code Since the approach promotes for an explicit mapping between architecture and code, the approach also supports to increase the awareness about how architecture is actually implemented in the source code.

Assessing the Decisions' Implementation After the Fact: The defined language can be understood as a means for architecture and code reviews in order to verify whether concepts have been consistently used. Additionally, *ArchCNL* can be also used for the tool-supported validation of architecture rules as shown in Section 8.4.

8.5.3. Critical Discussion of the Approach

In this section, potential limitations and challenges of the proposed approach are discussed.

Required Effort to Formalize and Maintain the Architecture Concept Language

First, a certain amount of effort is needed in order to create the architecture concept language and the architecture-to-code-mapping. However, this effort pays off in several ways. The architecture concept language helps to create and preserve a common language and understanding about the software architecture within the team. This is similar to the idea of DDD [Eva04], where developers and domain experts aim to achieve a joint understanding about the concepts that are used in the business domain and to make those concepts explicit in the source code. Such a mindset might also be beneficial for software architecture. This can be supported by the approach. Additionally, since the ontologies can be organized in modules, concepts and corresponding rules can be refined and reused throughout other projects.

In this way, the effort for defining a new architecture concept language can be reduced. For example, a project-independent architecture language that captures architecture concepts like *Repository*, *Entity*, *Aggregate Root*, and *Bounded Context* from DDD can be defined. This language and the corresponding rules could be reused by other projects that also implement the DDD approach in order to describe the architecture and to validate whether the code conforms to the design rules of DDD. In the context of Service-Oriented Architectures (SOA), concepts like component, service, or enterprise service bus can be captured in an architecture concept language and be reused in projects that also adopt SOA patterns.

Second, the maintenance and evolution of the architecture concept language may be challenging. During the evolution of the software system, there could be the need to add new architecture rules to be checked. For this, a software architect or developer who originally did not create the ontology might add new concepts. It is necessary to make sure that he does not introduce concepts to the existing language that are already defined with a different name. Decisions about new concepts and rules need to be thoroughly discussed within the team in order to mitigate this risk.

Challenges of Flexibility

Due to the flexibility of *ArchCNL* to express any concept and relation, it addresses and resolves the limitation of existing approaches that are restricted to predefined, inflexible meta models. As a consequence, these approaches are not able to formalize all architecture rules as shown in Section 8.2.7. However, this flexibility may also represent a challenge in architecture rule formalization. Since concepts and relations can be defined arbitrarily and are not restricted to predefined ones, any rules can be expressed that are not architectural per se or do not even belong to the software engineering domain. For example, one might express the following rule: `Every Cat must eat a Mouse.` Although this is a syntactically-correct sentence, this is not the intended use. In its current implementation, *ArchCNL* does not provide any mechanism to distinguish architecture rules from "non-architecture" rules. The approach leaves the decisions to the software architect whether a rule is an architecture rule or not. This is intended to a certain extent, since only the architect can decide if a rule should be considered as an architecture rule.

There is also a risk that architects define concepts and relations that are difficult to be mapped to the source code, since the approach does not provide a mechanism to restrict the selection of appropriate concepts and relations. As a matter of fact, rules have been identified in the projects that seem to be hard to be verified against the source code due to this problem. For example, the following rule has been defined by the software architect of project 1: *Resource-intensive analyses (e.g., call graph calculation and dependency analysis) should be performed only once.* The main challenge here is to define an appropriate mapping in order to identify code parts that implement the concept *resource-intensive analysis.* As opposed to concepts and relations defined in other rules, the concept *resource-intensive analysis* has no explicit counterpart in the source code. Since the mapping is not clear, using this rule for automatic architecture conformance checking becomes challenging. This means, the software architect needs to carefully choose architecture concepts and relations that can be clearly mapped to the code, so that the architecture rule can be verified in the implementation.

In order to approach the challenges described above, more examples of architecture rules need to be collected from practice, e.g., by following a study procedure as presented in Section 8.2. This may support software architects to better characterize architecture rules. For example, architecture rules can be evaluated based on whether defined concepts and relations can be explicitly mapped to the source code. This in turn can help software architects to refine their rule formalization, so that the rule can be verified in the implementation. In order to support software architect in this challenging process, a recommendation system could be provided that suggests appropriate concepts and relations based on a database containing a collection of empirically-grounded architecture rules.

Required Expert Knowledge for Writing ArchCNL

ArchCNL has been developed in a way that software architects and developers do not need to know about the underlying formalism in order to define the language and the architecture rules. *ArchCNL* hides the formalism used to formalize the language and the rules. However, when writing the rules, it could be helpful to know the basics of the description logic formalism, especially when language modeling errors are revealed by a consistency check. This still requires an expert who is able to understand and to resolve the error. In order to support the expert in writing the rules, the correct grammar of *ArchCNL* sentences can be enforced by a predictive editor that integrates the grammar of *ArchCNL*. Nevertheless, a developer or architect who only needs to read the rules does not need to understand the underlying formalism, since sentences in *ArchCNL*, i.e., architecture rules, do not contain any constructs that are specific to the formalism, and consequently, architecture rules read like natural language sentences.

Required Effort for Architecture-To-Code-Mapping

The approach assumes that architecture concepts and relations can be systematically mapped to code concepts. In a green field project, defining such a mapping might present only a few difficulties, since the initial concepts and relations are known and the complexity of the implementation is manageable. However, in brown field projects, the mapping might be more challenging. Architecture concepts, relations, and rules might be unknown and the actual mappings to the code are not clear, since original architects and developers of the software system have left the team. Additionally, the implementation of an architecture concept might be scattered over several code elements in the code base. This might complicate the definition of an architecture-to-code-mapping. Nevertheless, the approach allows software architects to map architecture concepts and relations to concrete instances of the code model instead of using conventions. Consequently, a mapping between architecture and code is still possible in brown field projects.

9. Conclusion and Future Work

This chapter summarizes the contributions of this thesis and discusses possible future work.

9.1. Contributions

This thesis presents a novel approach for supporting architecture enforcement by providing a method that allows for a flexible definition of a project-specific language to describe the software architecture and to formalize architecture rules. This language is called *architecture concept language*. For defining the architecture concept language, the approach applies description logics and ontologies. Moreover, the approach supports the automatic validation whether the implementation conforms to the architecture rules defined by the architecture concept language. It unifies source code artifacts into an ontology-based representation. Therefore, it allows architecture conformance checking to be performed on heterogeneous types of source code artifacts, e.g. Java source code, Maven artifacts, and Git history. Additionally, in order to provide a more understandable and usable notation, the approach includes *Architecture Controlled Natural Language (ArchCNL)*, a CNL software architects use to define the architecture concept language and corresponding architecture rules as natural language sentences. Having this type of specification language paves the way for a verifiable architecture documentation that is usable by and understandable for a large group of persons. Users of ArchCNL do not have to be experts in formalisms like description logics and ontologies. The approach has been proven to be applicable to large systems and in the context of industrial projects. In the following, the concrete contributions of this thesis are described in more detail.

9.1.1. An Empirical Study on Architecture Enforcement Concerns and Activities

The first goal of the thesis was stated as follows:

Goal: G1

Identifying the major concerns and activities of a software architect in the architecture enforcement process.

In Chapter 3, an empirical study on the current state of the practice of architecture enforcement has been presented. In this study, 17 experienced software architects from 16 different companies have been interviewed. The transcribed interviews have been analyzed with qualitative analysis and methods from grounded theory. The results of the study reveal enforcement concerns and activities. Additionally, the relationships between concerns and activities have been discovered in order to investigate which activities are performed for which concern. The observations of the study constitute the main drivers for a novel approach. It was found that it is crucial to establish a common language about the architecture and to make it visible in the source code.

9.1.2. Ontology-Based Architecture Enforcement

The approach developed in this thesis is based on the two goals:

Goal: G2

Developing an approach that allows for a formal, flexible, and understandable definition of the project-specific language used by software architects and developers in order to support establishing a common understanding of the software architecture. The approach helps to define a formal meaning of the terms used in the context of the project.

Goal: G3

Developing an approach that allows for the automatic validation of the project-specific language against the source code of the software system. The approach shall help to find indications in the source code where the constraints of the language have been violated, so that the software architect can take corresponding actions to address the violations appropriately.

In Chapter 6 and Chapter 7, a novel, flexible approach for architecture enforcement has been presented. The presented approach is based on the fact that the architecture concept language can be formalized with support of description logics and ontologies. Architecture concepts used in the conceptual architecture map to concepts of the description logics formalism, whereas relationships between architecture concepts can be mapped to roles of description logics. Consequently, architecture rules that describe the meaning of architecture concepts can be mapped to class axioms. Reasoning services are then applied to verify architecture rules for consistency against the implementation. ArchCNL as part of the approach is an understandable specification language used to define the architecture concept language. The grammar of ArchCNL is based on predefined architectural rule types. These are defined based on the semantics of concept constructors provided by the underlying description logics formalism. Each rule type has a corresponding predefined sentence structure in ArchCNL. Sentences written in ArchCNL are automatically transformed to class axioms. ArchCNLCheck as the second contribution of the approach implements ontology-based architecture conformance checking. It implements a process, where source code artifacts are automatically transformed to ontology-based representations and then verified against class axioms in order to reveal architecture violations. Architecture violation results are stored together with the rules and the source code artifacts in a unified knowledge base.

9.1.3. An Empirical Evaluation of the Approach

In Chapter 8, an evaluation of the approach is presented. The evaluation validates the flexibility of ArchCNL, its applicability in terms of understandability, naturalness, and usability, and the violation detection quality of ArchCNLCheck. Additionally, it is discussed how the entire approach (comprising ArchCNL and ArchCNLCheck) integrates into the enforcement activities discovered in the study in Chapter 3. The approach has been proven to be flexible. This means it is able to formalize a great amount of architecture rules from three industrial projects. The architecture rules show a diverse set of characteristics, i.e., architecture rules are not only based on static dependency rules. The approach is therefore more flexible than existing approaches that mainly focus on static dependency rules. Based on a focus group performed with 12 experienced software engineers, the approach was evaluated in terms of its applicability, i.e., perceived understandability of ArchCNL, the perceived naturalness of the rule formalization, and the perceived usability of ArchCNL. The evaluation shows promising results for these evaluation criteria and practitioners believe that the approach is useful. Additionally, the evaluation revealed interesting points for further improving the approach.

In terms of architecture detection quality, ArchCNLCheck is able to compete with existing approaches. Due to the flexibility of the approach, it is possible to formalize and validate architecture rules that cannot be verified by existing tools. That is why, it is able to find crucial architecture violations that remain undetected with the reference tool that was used for the comparison.

9.2. Future Work

In the following, several potential directions for further research are presented.

Extended Evaluation: In the industrial evaluation, only the perceived applicability has been evaluated. In order to evaluate the actual applicability (i.e. understandability, naturalness, usability), practitioners could actually apply the approach in their project. Additionally, more case studies should be conducted in order to improve the general validity of the violation detection quality.

Integration into the Software Engineering Process: The approach is implemented as a toolchain in order to automate the conformance checking process. The toolchain can be extended in such a way that it can be integrated in continuous integration pipelines. In this way, architecture rules can be checked continuously every time a developer submits a code change to the version control system that hosts the source code.

Additionally, an appropriate process model could be defined that integrates the approach into the development process applied by a development team. For example, this process model defines when and how the architecture concept language is defined in the development process and who is responsible for maintaining and evolving the architecture concept language and its corresponding documentation in ArchCNL.

Semi-Automatic Extraction of Architecture Concept Languages: Software architects and developers could be provided with a semi-automatic mechanism that recommends an architecture concept language based on several resources created during the architectural synthesis phase or during programming. Using approaches from information retrieval [BL10] and natural language processing [Cho03], e.g., named entity recognition from natural language text [NS07] or even from source code, appropriate architecture concepts and relations could be recommended. The algorithm could additionally recommend architecture rules for architecture concepts found in these sources.

Quality Model for Architecture Concept Languages: In the current implementation of the approach, the user does not get any feedback on the suitability and quality of the architecture concept language. It would be helpful to provide a quality model for the architecture concept language enhanced with appropriate metrics in order to evaluate the quality of the architecture concept language. The quality model could provide information about the structural complexity of the architecture concept language. As it was found in the study in Chapter 8, structural complexity greatly impacts the understandability of the architecture concept language.

Getting valuable feedback on the structural complexity could help architects and developers to further improve the language. Another quality aspect that could be measured may be the conciseness that measures whether the language contains redundant elements. A quality model could also include a measure for the completeness of the language, i.e., whether all important concepts are covered by the language. The concepts used in the language could be compared with entity and relation names used in architecture documentations. For this, natural language processing techniques could be applied.

Architecture Violation Repair: When architecture violations have been detected, the next task is to resolve them. However, when the number of findings is huge, an appropriate means for refactoring guidance is necessary. This could be supported by a process model or even a semi-automated approach that suggests an optimal sequence of refactorings in order to resolve the violations. An interesting approach is presented in [HM14] that is based on search-based refactoring [OC08]. It proposes optimal refactoring sequences that consist of move refactoring operations. The goal is to resolve violations by moving code elements with a high amount of violations to the most appropriate module in order to reduce the overall number of architecture violations in the implementation.

9.3. Closing Remark

There is no doubt that architecture enforcement is a complicated process that can only be managed appropriately by effective methods and tool support. The solution proposed in this thesis aims to support software architects in their key responsibility and to help developers to better understand the software architecture. In this thesis, architecture enforcement is understood as a process of establishing and explicitly capturing a shared architecture concept language in the team to increase the architectural awareness and consensual agreement on used architecture concepts and relations. This understanding of architecture enforcement is based on the viewpoint of "*software architecture as a language*" [Vö10] [Smo02] and not only as a "*set of structures*" [BCK12]. Architecture enforcement additionally requires to regularly verify whether the language is consistently and correctly used by the developers in the source code. However, for software systems with up to several hundred million lines of code, ensuring the conformance to the language manually is time-consuming and error-prone. That is why, tool-supported analyses are a prerequisite. For this, formal approaches are a necessity. However, they lack an understandable mechanism that is accepted by architects and developers and integrate in their toolchains. As shown here, the software engineering community can greatly benefit from formalisms like description logics and ontologies. CNLs - and ArchCNL in particular - are a promising direction to integrate those formal approaches even better into software development processes, so that they will become a part of everyday lives of architects and developers and to provide an understandable, living, and verifiable architecture documentation.

Appendices

A. Architecture Enforcement in Practice

A.1. Interview Guide

In the following, the interview guide with the main questions is presented. As the interview is designed as a semi-structured interview, it is possible that additional, context-dependent questions were asked. Those questions are not shown here. Moreover, the order of the guiding questions differed from interview to interview.

A.1.1. Part I: Briefing/Participants' Background

- Please, describe a recent project you are working for or you have worked for.
- What type of application is developed/maintained in this project?
- What are the important architecture decisions that were made?
- How large is the development team and the architect team, respectively?
- How many years of experience do you have as an architect?
- Do you have other duties in the project beside software architecture related tasks? If yes, which types of duties?

A.1.2. Part II: Enforcement Concerns

- Regarding the most important architecture decisions in the project, which aspects and concerns of those decisions are most important and should be definitely followed by the implementation?
- What are typical/critical/recurring problems concerning those decisions (in the code)? Or more specifically which types of architecture violations do you often see in the implementation?
- In case you participate in source code reviews, on which source code aspects that are important regarding architecture do you focus especially?

A.1.3. Part III: Enforcement Activities

- How do you ensure that your architecture and your concerns are implemented as intended? Do you follow any strategies?
- Are you involved in implementation task or are you working close to the implementation?

- In case you participate in source code reviews, what are the specific steps you perform when you inspect the source code in order to assess the implementation of the architecture decisions?
- Do you define formal rules for architecture? What type of rules are they? Do you check them automatically?
- How do you use architecture documents and models during the enforcement process?

A.2. Coding and Concepts List

In the following sections, the codes for each category of architecture enforcement concerns and activities are presented in Table A.1 and Table A.2. These codes emerged from the interview transcripts of the empirical study presented in Chapter 3. Based on these codes, enforcement concerns and activities have been derived.

A.2.1. Architecture Enforcement Concerns

Table A.1.: *Associated codes for architecture enforcement concerns.*

	Design Decisions
Aligning with Pattern Characteristics	**keypoint**: checking layer architecture automatically, **keypoint**: enforcing layering, **keypoint**: no business logic in frontend, **keypoint**: only component access via its api, **keypoint**: teaching architecture patterns new to developers, **keypoint**: dependencies between layers, **keypoint**: controlling architecture pattern implementation (mvc), **keypoint**: layer architecture, **keypoint**: ensuring layer architecture, **keypoint**: no clear responsibility of component, **keypoint**: layer violations, **keypoint**: no data conversion in business layer or frontend, **keypoint**: enforcing the most important patterns in implementation, **keypoint**: layer violations more likely if more layers, **keypoint**: no data mapping objects in code, **keypoint**: enforcing communication style, **keypoint**: finding pattern unrelated code, **keypoint**: enforcement of patterns on high level

Ensuring Architecture Design Principles	**keypoint**: avoiding coupling and synchronous communication, **keypoint**: complex dependencies between classes, **keypoint**: checking appropriate assignment of functionality to layer, **keypoint**: favouring loose coupling between components via asynchronous communication, **keypoint**: correct separation of components is essential for deployment, **keypoint**: usage of types in wrong component, **keypoint**: unwanted dependencies between components, **keypoint**: evaluating separation of concerns, **keypoint**: evaluating modularization, **keypoint**: developers build monolithic structures, **keypoint**: wrong assignment of responsibilities to components, **keypoint**: violating separation of concerns, **keypoint**: inappropriate separation of components makes deployment difficult, **keypoint**: evaluating package coupling, **keypoint**: enforcing component boundaries, **keypoint**: evaluating and controlling modularization, **keypoint**: dependency cycles, **keypoint**: ensuring a component-oriented code structure from the beginning, **keypoint**: dependency violation severity, **keypoint**: separation of application and technology architecture, **keypoint**: evaluating dependencies, **keypoint**: minimizing dependencies for increasing comprehensibility, **keypoint**: dependency violations, **keypoint**: avoiding cyclic dependencies, **keypoint**: avoiding too many and complex dependency structures

Differentiating between Macro and Micro Architecture Decisions	**keypoint**: architect needs to consider strategic and tactical view, **keypoint**: architect responsible for high abstraction level, **keypoint**: architect responsible for macro architecture, **keypoint**: characteristics of global view of architecture, **keypoint**: each development team decides for its microarchitecture, **keypoint**: developers are architects on tactical level, **keypoint**: global = strategic, **keypoint**: global view of architecture, **keypoint**: ensuring macro architecture, **keypoint**: architecture work also in the code details, **keypoint**: microarchitecture, **keypoint**: microarchitecture as developer's responsibility, **keypoint**: differentiating between architecture and programming aspects, **keypoint**: developers responsibility on low abstraction level, **keypoint**: strategic vs. tactical architecture view, **keypoint**: differentiating between two views of architecture, **keypoint**: macro architecture, **keypoint**: decisions on code-level are not evaluated/verified, **keypoint**: macro architecture as general architecture of system, **keypoint**: maintaining big picture of architecture, **keypoint**: architect responsible for strategic view, **keypoint**: architect taking care of microarchitecture, but no enforcement

	Implementation Quality
Appropriate Use of Technology	feature variety of technologies/practices, consequence of tool dependencies, guideline - messaging model (topics vs. queues), guideline - no automatic generation of database scheme, guideline - no RMI in Java, guideline - only asynchronous communication, enforce messaging model, enforce technology kind-of-use, **keypoint**: technology use (impact), **keypoint**: restricting technology features in favor of testability, **keypoint**: thoroughly considering non-functional requirements before choosing technology, problem - tool dependencies, **keypoint**: thoroughly selecting technology features that are really needed, problem - technology overuse, problem - technology has drawback on quality, **keypoint**: controlling technology use, problem - technology can violate architecture principles, technology prevents A&T separation, **keypoint**: dealing with inappropriate tooling, problem - aim for technologies, **keypoint**: dealing with manifold features of technologies, **keypoint**: unnecessary use of technology, **keypoint**: enforcing specific type of use of technology, aim for tools, **keypoint**: lightweight use of technology, **keypoint**: flexibility of technologies, **keypoint**: developers being biased by specific technologies, **keypoint**: missing understanding of technology's impact on architecture, **keypoint**: focusing on technology rather on the software system's functionality/requirements, **keypoint**: unwanted dependencies due to tools, **keypoint**: forcing use of technologies with no reasons impacts architecture badly, **keypoint**: usage of complex tools leads unknowingly to violations, **keypoint**: tool dependencies, **keypoint**: framework restrict architecture, **keypoint**: frameworks can decrease architecture quality

Visibility of Domain Concepts in Code	concern - terminology, guidelines - naming conventions, violation - namings, appropriate terminology enables comprehensibility, appropriately aligning components with respect to business domain, guideline - define domain oriented data types, **keypoint**: domain-driven namings, **keypoint**: favoring classes driven by business domain concepts, **keypoint**: mapping business domain concepts onto OO-classes, **keypoint**: communication between components via domain.oriented objects, **keypoint**: enforcing domain-oriented naming conventions, **keypoint**: disliking primitive datatypes in interface of business logic code, **keypoint**: evaluating the alignment of component structure with business domains, guideline - domain oriented data model, guideline - use domain objects
Visibility of Architecture in Code	making architecture visible in code, aligning code structure with architecture, aligning namings in code with terms used in architecture, **keypoint**: easily locating architecture in code, architecture-code-mapping, technical terminology and structural indicator, **keypoint**: enforcing/encouraging that architecture is visible in code
Code Comprehensibility	appropriate terminology enables comprehensibility, **keypoint**: architect being responsible for code comprehensibility, **keypoint**: code comprehensibility as a main concern, **keypoint**: minimizing dependencies for increasing comprehensibility, **keypoint**: naming conventions (comprehensibility), **keypoint**: naming conventions (layers), **keypoint**: orientation in code (comprehensibility), code comprehensibility supports architecture conformance, **keypoint**: evaluating code structure (code review)
Ensuring and Verifying Runtime Quality	**keypoint**: checking code with respect to ASRs, **keypoint**: code review driven by architecture significant requirements, **keypoint**: enforcing security related rules, **keypoint**: evaluating implementation based on non-functional requirements

A.2.2. Architecture Enforcement Activities

Table A.2.: *Associated codes for architecture enforcement activities.*

Achieving Mutual Understanding

Activity	Codes
Modelling Software Architecture for Developers	common understanding about architecture, documentation type: arc42, **keypoint**: missing knowledge of developers due to documentation, **keypoint**: models for general overview, **keypoint**: comprehensive documentation not supportive during development, **keypoint**: models for helping new developers in understanding system, **keypoint**: models on component level and interaction between components, **keypoint**: models on high abstraction level, **keypoint**: models supporting during implementation, documentation type: FMC, documentation type: UML, architecture in the head, encourage mental model about architecture, **keypoint**: using component diagrams, **keypoint**: sequence diagram for communication between systems, **keypoint**: avoiding fine-grained models, **keypoint**: only documentating the most important things, **keypoint**: favouring drawings that are easy to understand, **keypoint**: using drawings to explain the general structure of architecture, **keypoint**: using models as development template, **keypoint**: using documentation for efficient communication, **keypoint**: maintaining big picture of architecture, common mindset, developer needs to understand the concept, **keypoint**: preferring living documentation, **keypoint**: using models for many people/stakeholders, **keypoint**: low effort for documentation (whiteboard), **keypoint**: using models for visualization,

Achieving Mutual Understanding	
Activity	Codes
Modelling Software Architecture for Developers	**keypoint**: encouraging developer's architecture understanding, **keypoint**: using simple drawings for architecture models, **keypoint**: using models for long-living structures, **keypoint**: wiki based documentation, **keypoint**: using models for comprehension, **keypoint**: preferring pictures over comprehensive documentation, encouraging architecture understanding, **keypoint**: presenting developers the architecture documentation, **keypoint**: developers having the same mindset regarding architecture, **keypoint**: using whiteboard documentation, **keypoint**: architecture diagrams rather used for explaining the system than for verification, **keypoint**: architecture documentation must be practicable, **keypoint**: documentation as guide not as a checklist, **keypoint**: documentation must be applied consistently, **keypoint**: make architecture model accessible, **keypoint**: verifiable documentation
Ensure Feasibility of Architecture	
Gathering Feedback	communication - discussion about solution, communication - discussion about violations, evaluation - find rationale/causes/reasons for violations, practice - involvement of developers, practice - involvement of experience developers, decision of developer might be better, **keypoint**: architect encouraging regular communication between architects and developers, **keypoint**: designing architecture rules together with experienced developers, determine architecture rules together, determine programming rules together, **keypoint**: presenting violations to developers, **keypoint**: being open for feedback, **keypoint**: tight collaboration with development team, **keypoint**: getting feedback on strategic architecture from developers, regular meeting with developers, **keypoint**: preferring discussions with developers for finding best solutions (no strict rules), **keypoint**: discovering guidelines together, **keypoint**: developer does not accept architecture, **keypoint**: architect respects developer's preferences regarding rules, **keypoint**: feedback from developers regarding architecture is essential, **keypoint**: challenges in convincing developers regarding architecture

Revising the Architecture	adapt architecture, **keypoint**: architect being able to change the original architecture, **keypoint**: architect being open for revising architecture, **keypoint**: evolving architecture, **keypoint**: changing things gradually, **keypoint**: evolving architecture incrementally, **keypoint**: preferring being a coach rather than enforcing strict formalized rules, **keypoint**: communicate architecture changes quickly with the team, **keypoint**: cleaning up architecture, **keypoint**: revising architecture rules since code too complex, revising architecture, solution cannot be implemented, **keypoint**: rules not always appropriate for given situation
Adjust Architecture to Skills	assess developers' knowledge and skills, use patterns known by developers, **keypoint**: defining uniform programming guidelines due to varying skills in development team, **keypoint**: architecture skills of developers, **keypoint**: developers having architecture skills, **keypoint**: missing knowledge about important architecture concepts, **keypoint**: using architectural solutions known by developers, programming habits and experience of developers, **keypoint**: minimum number of developers having architecture skills, **keypoint**: missing understanding about architecture concepts, **keypoint**: architect knowing the individual skills of developers, **keypoint**: developers need architecture knowledge/skills, **keypoint**: basic competence needed by developers, **keypoint**: prefer well-known solutions over complex architecture design, **keypoint**: knowing individual experience and knowledge level in development team, personal quality, **keypoint**: developer's programming habits, **keypoint**: unknown concepts in development team (value objects), **keypoint**: patterns understood and known by developers are rarely violated, assess developers knowledge and skills, developer's skills and experiences

Providing Implementation Templates	
Architectural Skeletons	architectural templates - properly created according to decisions, **keypoint**: architect building prototypes, practice - architectural templates, practice - architectural templates as implementation guide, **keypoint**: building architecture templates as guiding implementation for developers, **keypoint**: supporting developers by building prototypes, **keypoint**: building proper prototypes following best practices, **keypoint**: motivate architecture based on architectural templates, **keypoint**: supporting developers with code templates, **keypoint**: providing supporting structures for developers, **keypoint**: architecture templates as reference for code reviews, practice - templates as reference for reviews, **keypoint**: code templates as architecture blueprint, practice - architectural prototype
Code generation	practice - code generation, **keypoint**: generating code templates from documentation, **keypoint**: generating database schema
Awareness of Architecture in Code	
Correlate Architecture and Code	architecture-code-mapping, **keypoint**: developers not aware of the impact of code changes on architecture, **keypoint**: drift between architecture documentation and code, **keypoint**: enforcing developer's awareness regarding impact of coding modifications on architecture, **keypoint**: mapping layers and slices to packages, **keypoint**: architecture guiding code, **keypoint**: naming conventions (layers), **keypoint**: good mapping between architecture and code helps code orientation

Assessing the Decisions' Implementation after the Fact	
Code Review	using tools for finding hotspots, **keypoint**: architect conducting code reviews, **keypoint**: code inspection is necessary for evaluation of architecture quality, **keypoint**: code review as software developers responsibility, **keypoint**: regular peer reviews, **keypoint**: code review investigating component structure, **keypoint**: architecture workshop (sotograph), **keypoint**: architecture workshop (when?), **keypoint**: using tools for code smell detection, **keypoint**: code review driven by architecture significant requirements, **keypoint**: communication and reviews as mechanisms for finding architecture problems, **keypoint**: conducting code reviews (number of persons involved), **keypoint**: code review during pair programming, **keypoint**: evaluating metrics with tools, **keypoint**: evaluating programming and architecture rules during code review, **keypoint**: investigating static dependencies, **keypoint**: using code analysis results as starting point
Repository Mining	**keypoint**: finding architecture violations by reviewing commits, **keypoint**: gerrit code review
Model-Code-Comparison	**keypoint**: comparing models with actual implementation, **keypoint**: comparing planned models with reverses engineered models, **keypoint**: generating documentation/diagrams from code (aim), **keypoint**: generating models from code for general overview of implemented structures
Testing	test coverage, mapping requirements to test, automated tests, tests as documentation, test coverage supports architecture conformance, **keypoint**: automated tests for ensuring fulfillment of functionalities

B. Literature Review on Architecture Enforcement

A literature review for finding evidence about architecture enforcement in current literature has been conducted. It has been investigated to which extent the codes and categories that emerged from the interviews are covered in the literature. It is worth noting that the literature review is conducted after the interviews. However, the literature was searched independently from the findings of the interviews. The codes and categories emerged from the interview results have not been used in order to formulate the search queries for finding relevant literature. This ensures that as many references as possible about architecture enforcement can be found. During the analysis of the literature dataset, the codes and categories from the interviews are used to label phrases in the selected publications. In this way, it is investigated which codes and categories are also covered in literature. No new codes and categories have been derived from literature, this means, all findings (codes and categories) emerged from the analysis of the expert interviews as the aim of the study is to investigate architecture enforcement in practice.

The main goal of the review is to focus on empirical studies on enforcement concerns and activities. Practices from the systematic literature review (SLR) process as suggested by [KC07] have been adopted to make the literature search credible, unbiased, and reproducible. The goal of the review was to find relevant literature that potentially considers and discusses software architect's concerns and activities during architecture enforcement. In the following sections, the necessary steps of the review to collect relevant literature are described, namely 1) defining the search process, 2) defining inclusion and exclusion criteria, 3) the data collection procedure and finally 4) the data synthesis.

B.1. Search Process

A search strategy has been developed for the literature search that includes 1) a search query term, 2) the relevant sources where the publications will be collected from and 3) defining the parts of the publications which should be searched (title, abstract, full text etc.).

As *architecture enforcement* is not a common term used in software architecture literature, this term needs to be defined indirectly in the search query. For this, appropriate synonyms to paraphrase *architecture enforcement* or *enforce* (e.g. manage, force, guide, lead) and likewise the terms *concern* (e.g. interest, consider, worry, need, wish) and *activity* (e.g. role, duty, skill, care, task) have been used. Additionally, the terms *software architect* and *developer* (and corresponding synonyms) have been added to the query in order to be able to find publications that discuss or consider the relationship between the both, as it is important for architecture enforcement. This then resulted in the following search query:

software architect **AND** (developer **OR** programmer **OR** code **OR** implement) **AND** (role **OR** skill **OR** concern **OR** interest **OR** consider **OR** worry **OR** task **OR** duty **OR** matter **OR** role **OR** need **OR** wish **OR** demand **OR** urgency **OR** essential **OR** important **OR** care

OR activity **OR** collaborate **OR** communicate **OR** guide **OR** teach **OR** coach **OR** lead **OR** manage **OR** enforce **OR** force)

The Boolean AND operator was used to connect the search terms in order to narrow the search. The Boolean OR operator was used to enable alternative terms and a wider range of search results. It is worth mentioning that not all search engines of digital libraries allow such long search queries (e.g. IEEExplore only allows 15 terms in a query). That is why, there is no need to split the query into several sub-queries. The literature is collected for each sub-query. The datasets of the queries were compared in order to find duplicates. The datasets were then finally merged into one dataset. To increase the likelihood of finding relevant data sources, the target of the search query was defined to be applied to the full text and meta data. The search has been applied using the query on four well-known digital libraries: SpringerLink, IEEExplore, the ACM Digital Library, and ScienceDirect.

B.2. Inclusion and Exclusion criteria

In order to decide whether a publication should be added to the result set, the SLR method requires to explicitly define inclusion and exclusion criteria. They are listed in Table B.1. They assess the fitness of the content in each potential relevant publication to the defined research questions. They are applied after all full texts have been retrieved. The inclusion or exclusion of publications has been decided based on the title and the abstract. In case it is not clear whether to include the publication, it has been added to the data set for further investigation. In a second phase, the full text of those publications was read carefully in more detail. If relevant information could be found (interests, activities of the architects etc.) the publication was added to the final result data set. Publications have been included that discuss the role of the software architect and that could be of particular interest in the context of architecture enforcement. Those publications can be empirical studies, but also experience reports from (former) software architects who provide insights in their work and/or give suggestions for other software architects regarding their architecture work. It has also been decided to include editorials into the dataset, if appropriate. They often provide valuable discussions about the role and interests about the software architect from an experienced architect's point of view (e.g. reports from John Klein, Frank Buschmann, Eoin Woods, Martin Fowler in the Pragmatic Architect Series like in [Bus09]), although the reported results are mostly not collected with a rigorous research method. Additionally, papers that only propose a new tool are not of interest. The focus of those papers is the description of new tools and the evaluation of their effectiveness, e.g. in finding architecture violations or recovering software architectures from code. Those publications do not discuss the role, concerns, and activities of the software architect.

Table B.1.: *Defined inclusion and exclusion criteria for selecting publications. The criteria are numbered. The prefix I designates an inclusion, the prefix E an exclusion criterion.*

Inclusion/Exclusion	Criteria
I1	Publication discusses the role of the software architect.
I2	Publications should define or discuss concerns and activities that could be related to architecture enforcement.
I3	Publications must be written in English.
I4	Publications published in journals, conferences, book chapters, magazines (also editorials), and workshops proceedings will be included.
E1	The publications such as abstracts, position papers, short papers, tool papers, poster summaries, keynotes, tutorial summaries, conference summaries (or introduction to conference proceedings), workshop summaries or panel summaries are excluded, since they do not provide a reasonable amount of information.
E2	Enforcement considered in the publication is not in the context of software architecture (e.g. policy enforcement, enforcement of law regulations and alike)
E3	Publication's full text is not available.
E4	The publication proposes a tool, e.g. for architecture conformance checking.

B.3. Snowballing

In order to not miss important publications, Snowballing has been performed afterwards. The suggestion of [Woh14] has been followed to apply backward and forward snowballing to increase the likelihood of finding more relevant publications. Snowballing is an iterative method that aims to harvest the references of a paper from the initial result set. Backward Snowballing collects the publications referenced in the reference list of the respective paper. Forward Snowballing identifies relevant papers based on the papers citing the paper that is currently investigated. All papers from the result set of the first stage are investigated in this way. Potential references are eventually added into the final result data set if they fulfill the defined inclusion criteria (see Section B.2).

B.4. Data Collection, Synthesis, and Analysis

In order to collect the necessary information about architecture enforcement concerns and activities, a deductive coding on the publications has been performed. The categories regarding concerns and activities that resulted from the interviews have been used as the coding scheme. AtlasTi has been used as a supportive tool, the same tool that was used for analyzing the interviews. Words, sentences, or paraphrases in the publications that can be related with one or more of the concerns and activities are labeled with a code that is named after them.

Table B.2.: *Mapping enforcement concerns to publications containing statements that were labeled with the corresponding code.*

Concern	**Literature**
Ensuring Architecture Design Principles	[McB04], [Lag14], [BSD16], [Bus11a], [CLN14], [SUS14], [Sau10], [PEE12], [MPB14], [SM17], [Kru99], [Woo15]
Appropriate Use of Technology	[McB04], [CLN14], [Bus10], [GCSY08]
Aligning with Pattern Characteristics	[SH15], [BSD16], [MPB14], [Woo15]
Visibility of Domain Concepts in Code	[BH10], [Bus12]
Visibility of Architecture in Code	[BH10]
Code Comprehensibility	[UD10], [CLN14]
Differentiate between Macro and Micro Architecture Decisions	[McB04], [CHS09], [CLvV07]
Adhere to Standards	[Ber08], [GKB10], [GCSY08]
Ensuring Runtime Quality	[CLN14]

As an example, the following statement was mapped to the activity *Adjust Architecture to Developers' Knowledge, Skills, And Experience*: *"The smartest design is basically useless if the development team lacks the skills to implement it properly. Pragmatic architects therefore take explicit care that they "design to skill." From all potential alternatives to resolving a specific aspect or concern, they choose the option with which the development team feels most habitable and familiar"* [Bus11c]. After analyzing all the selected publications in this way, the concerns and activities have been mapped to the publications that mention the respective aspect. The results are shown in Table B.2 for the enforcement concerns and in Table B.3 for the enforcement activities, respectively. Having this information, the interview results are compared with the findings from literature. This gives an impression to what extent the concerns and activities discovered from the interviews are covered in current literature and which evidence is missing there.

Table B.3.: *Mapping enforcement activities to publications containing statements that were labeled with the corresponding code.*

Activity	**Literature**
Modeling Architecture For Developers	[UD10], [AMK16], [TKF16], [CLN14], [MPB14], [Bus12]
Gathering Feedback	[McB04], [Kru08], [CHS09], [Kle05], [EP16], [GAMD16], [SUS14], [MPB14], [Kru99], [Fab10]
Revising the Architecture	[UD10], [Fab10]
Correlate Architecture and Code	-
Code Generation and/or Architectural Skeletons	[Ber08], [CH10], [BSD16], [EP16], [CLN14], [Fab10], [Woo15], [Bus12]
Adjust Architecture to Developers' Skills	[Kru08], [Kle16], [TKF16], [Sau10], [Bus11c]
Code Review	[SH15], [Ber08], [Kru08], [UD10], [BSD16], [CHS09], [TKF16], [MPB14], [Kru99]
Repository Mining	[UD10]
Model-Code-Comparison	-
Automatic Validation of Architecture Rules	[UD10], [Lag14], [CLN14], [MPB14]
Traceability	[AMK16]
Testing	[Ber08], [PZ10], [EP16], [Sau10], [MPB14], [Bus11c], [Bus11b]

C. Evaluating the Flexibility of ArchCNL: Interview Guide

In this section, the interview guide for the study presented in Section 8.2 is presented.

C.1. Background Information

General Experience and Background

1. How many years of professional experience do you have in software engineering/developing software?
2. For which domain do you develop software?
3. What kind of software systems do you develop?

Experience with Formalization Methods

1. Do you have experiences with formal methods?
    a) If yes, which ones? for example Behavior Driven Design, Architecture Description Languages, Object Constraint Language
    b) If you have experience, how much experience (in years, to which extent) do you have with formal methods?
2. What is your attitude regarding formal methods?
    a) Are they necessary, helpful, useless?
3. Do you have experiences with architecture rule formalization and validation tools (e.g. architecture conformance checking tools)?
    a) Have you ever used tools for architecture rule formalization and validation (in your projects)?
4. If yes, which one? (e.g. Sotograph, Sonarqube, ArchUnit, jQassistant...)
    a) Do you find them helpful?

Project Background

1. What is the size of the team?
    a) How many developers?
    b) How many architects?

2. What is the size of your software system you develop?
    a) Approximately how many lines of code?
    b) How many projects/components?
3. How is architecture documented in your projects? (wiki, word document, formal models, issue tracking system, plain text files, whiteboard, no documentation...)
4. Are architecture rules documented in your project?
    a) If yes, how? E. g. separate (word/text/...) document, wiki, issue tracking system, no documentation?
5. Do you formalize architecture rules in your project?
6. Do you use tools for validating architecture rules?
    a) If yes, which one? (e.g. sotograph, sonarqube, archunit, jqassistant...)
    b) If no: why not?

C.2. Questions Regarding the ArchCNL Formalization

For each rule, the following questions are asked:

1. How understandable is the formalization of the rule in *ArchCNL*?
    a) Perfectly understandable, largely understandable, partially understandable, largely not understandable, not understandable
    b) In case it is not understandable, why? what hinders the understandability?
2. How artificial/natural does the formalization appear to you?
    a) If it does appear artifical, why?
3. How well does the *ArchCNL* formalization reflect the original intention of the architecture rule?
    a) Are concepts and relations of the original rule well represented?
    b) If it does not reflect the rule well: why?
    c) How similar is the formalization with respect to the original one in natural language?

C.3. Overall Evaluation

1. In your opinion, how useful would it be having such a "natural" formalization of architecture rules that can also be validated?
2. would you use this approach for specifying and documenting architecture rules?
    a) Yes: how would you use it?
    b) No: why not? what is missing? what are the disadvantages?

3. Do you think developers would be more aware of the architecture rules when using this approach (because it is a rather intuitive description)?
4. Do you think the *ArchCNL* formalization can be potentially understood by all team members (architects and developers)?
5. Do you think the approach would support developers and architects to respect the architecture rules during implementation?
6. What do you think are the key benefits of this approach?
7. What do you think are the key drawbacks/disadvantages/challenges of this approach?
8. Which features should the approach provide additionally?
9. Is there anything else you would like to say/discuss/add?

D. Evaluation of the Applicability of the ArchCNL

The material listed in this section refers to the study presented in Section 8.3. This section presents the questionnaire used in the workshop as well as the architecture rules that have been presented during the workshop.

D.1. Questionnaire

D.1.1. Part I: Background

1. How many years of professional experience do you have in developing software?
2. For which domain do you develop software? (e.g. Enterprise, Automotive etc.)
3. What is the size of the development team? (number of developers, software architects,...)
4. What is the size of the system that you develop (LOC)?
5. Do you have experience with formal methods in software engineering (e.g. Behavior-Driven Design, Architecture Description Languages etc.)? If yes, which?
6. In case you have experience with formal methods, how many years of experience do you have with them?
7. How is your attitude regarding using formal methods in software engineering? (positive - rather positive - neutral - rather negative - negative)
8. How do you document architecture in the project? (e.g. no documentation, textual documentation, UML, Whiteboard, Wiki etc.)?
9. Do you validate the architecture conformance of the implementation?
10. In case you validate architecture conformance, which methods do you use? (manual code review, automated validation with dedicated tools etc.)

D.1.2. Part II: Architecture Rules

Question 1: How understandable is the meaning of the rule in *ArchCNL*?

1: Completely understandable.

2: Mostly understandable.

3: Partly understandable.

4: Mostly not understandable.

5: Not understandable.

Question 2: How difficult does it appear to me to formulate the rule in *ArchCNL*?

1: I can formalize the rule.

2: I can mostly formalize the rule.

3: I can partially formulate the rule.

4: I find this rule difficult to formulate.

5: I cannot formulate this rule.

Question 3: How natural does the formalization appear?

1: Very natural.

2: Natural.

3: Neither natural nor artificial.

4: Artificial.

5: Very artificial.

D.1.3. Part III: Overall Evaluation

For the following statements, the participants were asked to choose an answer on the following 5-point Likert-scale.

Possible Answers

1: Great approval.

2: Approval.

3: Neutral.

4: Little approval.

5: No approval.

General Project Information:

- Architecture rules (similar to the ones shown here) are documented in my software project(s).
- Architecture rules (similar to the ones shown here) are validated manually in my software project(s).
- Architecture rules (similar to the ones shown here) are validated automatically in my software project(s).
- An automatic validation of architecture rules is reasonable.

General Evaluation of the Approach

- The *ArchCNL* is well suited to be used for architecture rule documentation.
- The architecture rules documented in *ArchCNL* can be well understood by all team members.
- It is possible to formulate architecture rules in *ArchCNL*, so that they are similar to the architecture language used in the project.
- The documentation of architecture rules in *ArchCNL* represents a small additional effort.
- I can learn the *ArchCNL* in a short time.

Applicability of the Approach in My Project

- It is possible to document a lot of rules that are relevant for my project with the *ArchCNL*.
- The *ArchCNL*-based documentation would support developers and architects to know the most important architecture rules in the project.
- The automatic validation of *ArchCNL*-based rules would support developers and architects to follow the most important architecture rules in the project.

Open-Ended Questions

- What did you like most about the approach?
- Where do you see the biggest hurdle for using the approach in practice?
- How would you integrate the approach into your project or where would you document the rules in *ArchCNL*? (simple text file, architecture models, word document etc.)?
- Which aspects of the approach would you improve?

D.2. Architecture Rules and Their Formalization

Table D.1 depicts the architecture rules that have been presented in the focus group in order to evaluate the approach. Architecture rules are given in natural language (English) and in the *ArchCNL* on the next page.

Table D.1.: *Architecture rules and their ArchCNL-based formalization, the source of the rule, and the associated complexity group (CG) (as calculated in 8.3) as presented in the focus group.*

ID	Rule	ArchCNL	Source	CG
1	Controllers must use Services.	`Every Controller must use a Service.`	Layers [BMR+96]	Group 1
2	Services can only be used by Controllers or other Services.	`Every Service can only be-used-by a Controller or be-used-by a Service.`	Layers [BMR+96]	Group 2
3	Controllers are not allowed to access data access objects.	`No Controller can access a DataAccessObject.`	Layers [BMR+96]	Group 1
4	Controllers cannot be accessed by other entities.	`Nothing can access a Controller.`	Layers [BMR+96]	Group 1
5	Each host transaction component must only be included once in a module.	`Every HostTransactionComponent can be-included-in exactly one Module.`	[WB14]	Group 1
6	Host transaction components must not have outgoing connections.	`No HostTransactionComponent can depend-on anything.`	[WB14]	Group 1
7	The name of a batch bean must start with "Bat".	Every BatchBean must have-name equal-to "Bat*".	[isya]	Group 1

Continued on next page

Table D.1 – continued from previous page

ID	Rule	ArchCNL	Source	CG
8	The application core calls methods defined by the MBean. The other direction (Mbean reads from the application core) is not allowed, since this could cause performance problems.	`No MBean can call a Method that (is-defined-in an ApplicationCore).`	[isyf]	Group 1
9	The primary key of an entity can only consist of a single attribute. This also means that no composite keys are allowed for primary keys.	`No Entity can define a CompositePrimaryKey. Every PrimaryKey can consistOf exactly 1 Attribute.`	[isyd]	Group 1
10	If an application requires exceptions, then each application exception must implement one of the three predefined exceptions: BusinessException, TechnicalException, TechnicalRuntimeException.	`Every ApplicationException must implement a BusinessException or implement a TechnicalException or implement a TechnicalRuntimeException.`	[isyb]	Group 1

Continued on next page

Table D.1 – continued from previous page

ID	Rule	ArchCNL	Source	CG
11	All public methods in the controller layer must return API response wrappers.	`Every Method that (has-modifier equal-to "public" and is-located-in a Controller-Layer) must return an APIResponseWrapper.`	[arcb]	Group 3
12	The type of generation (i.e. the strategy) provided by the @GeneratedValue annotation that is defined for a field must be set to "AUTO".	`Every GeneratedValueAnnotation that (annotates a Field) must define a Strategy that (hasValue equal-to "AUTO").`	[isyd]	Group 2
13	Aggregates must provide a method for allowing access on an Aggregate Id that is defined in this aggregate so that it can be accessed by other entities.	`Every Aggregate that (defines an AggregateId) must provide a Method that (returns an AggregateId).`	Domain Driven Design [Eva04]	Group 2
14	Fields that represent a Logger must be declared with "private static final"	`Every Field that (represents a Logger) must have-modifier equal-to "private" and have-modifier equal-to "static" and have-modifier equal-to "final".`	[isyc]	Group 3

Continued on next page

Table D.1 – continued from previous page

ID	Rule	ArchCNL	Source	CG
15	In case an exception is thrown when calling a service method, this exception must be an exception that is part of the service interface. No exception of the application core is thrown.	`Every Exception that (is-thrown-by a ServiceMethod) must be a ServiceInterfaceException. No ServiceMethod can throw an Exception that (is-defined-in an ApplicationCore).`	[isyb]	Group 2
16	For every exception that is part of the exception facade, a correlation ID must be set. This id is provided by a specific annotation (“@LoggingContext” Annotation). This means that every method that is defined by the exception facade must be annotated with this annotation.	`Every Method (that is-part-of an ExceptionFacade) must be-annotated-with a LoggingContextAnnotation.`	[isye]	Group 2
17	Service methods can only have transfer objects or primitive data types as a return or a parameter type.	`Every ServiceMethod can only return a TransferObject or return a PrimitiveType. Every ServiceMethod can only declare a Parameter that (has-type a TransferObject) or declare a Parameter that (has-type a PrimitiveType).`	[Her11]	Group 3

Continued on next page

Table D.1 – continued from previous page

ID	Rule	ArchCNL	Source	CG
18	Classes that have a field annotated with @Payload should only be accessed by @Secured methods.	`Every Class that (declares a Field that (is-annotated-with a PayloadAnnotation)) can only be-accessed-by a Method that (is-annotated-with a SecuredAnnotation).`	[arcc]	Group 2
19	Stateless session beans cannot have state. This means, fields in this bean cannot be accessed outside the constructor or a method that is annotated with @PostConstruct of this session bean.	`Every Field that (is-defined-in StatelessSessionBean X) can only be-accessed-by a Constructor that (is-defined-in StatelessSessionBean X) or be-accessed-by a PostConstructMethod that (is-defined-in StatelessSessionBean X).`	[arcd]	Group 4
20	The first parameter of every Method defined in the Remote Bean interface must have the type "AufrufKontextTO" or "ClientAufrufKontextTO".	`Every Parameter that (has-position equal-to 1 and is-defined-by a Method that (is-defined-by a RemoteBeanInterface)) must have-type a AufrufKontextTo or have-type a ClientAufrufKontextTo.`	[isye]	Group 3
21	Applications can only access components of the service layer of another application.	`Every Application that (accesses Application X) can only access a ServiceLayer that (belongs-to Application X).`	[WB14]	Group 3

Continued on next page

Table D.1 – continued from previous page

ID	Rule	ArchCNL	Source	CG
22	Services within one module must communicate via local EJBs.	`Every Service that (has-name equal-to A and is-located-in Module X and uses a Service that (has-name equal-to B and is-located-in Module X)) must communicate-via a LocalEJBCommunication that (has-source a Service that (has-name equal-to A) and has-destination a Service that (has-name equal-to B)).`	[WB14]	Group 4
23	Services of different domains must communicate via web services.	`Every Service that (is-located-in Domain X and uses a Service that (is-located-in Domain Y)) must use a WebserviceCommunication that (has-source a Service that (is-located-in Domain X) and has-destination a Service that (is-located-in Domain Y)).`	[WB14]	Group 4
24	Services within one application (but in different modules) must communicate via local EJBs.	`Every Service that (is-located-in Application A and is-located-in Module X and uses a Service that (is-located-in Application A and is-located-in Module Y)) must communicate-via a LocalEJBCommunication that (has-source a Service that (is-located-in Application A and is-located-in Module X) and has-destination a Service that (is-located-in Application A and is-located-in Module Y)).`	[WB14]	Group 4

Continued on next page

Table D.1 – continued from previous page

ID	Rule	ArchCNL	Source	CG
25	Business logic services must only use data access components, host access components and host transaction components if they belong to the same module.	`Every BusinessLogicService that (is-located-in Module X and uses a DataAccessComponent) can only use a DataAccessComponent that (is-located-in Module X) or use a HostAccessComponent that (is-located-in Module X) or use a HostTransactionComponent that (is-located-in Module X).`	[WB14]	Group 4
26	Different applications should communicate via web services or remote EJBs.	Every Application that (has-name equal-to X and communicates-with an Application that (has-name equal-to Y)) can only use a WebserviceCommunication that (has-source an Application that (has-name equal-to X) and has-destination an Application that (has-name equal-to Y)) or use a RemoteEJBCommunication that (has-source an Application that (has-name equal-to X) and has-destination an Application that (has-name equal-to Y)) .	[WB14]	Group 4

E. FAMIX Ontology

In this section, the detailed FAMIX ontology is given using the functional-style syntax. The transformation of the FAMIX ontology is performed based on the defined transformation rules given in Table E.1. First, an exemplary transformation is shown. In a subsequent section, the entire FAMIX ontology is listed.

E.1. Transformation of the FAMIX Meta Model to an Ontology-Based Representation

In this section it is illustrated how the meta model is translated into the FAMIX ontology by applying the transformation rules presented in Section 7.2.4. The translation is illustrated using the excerpt from Figure 7.4. The FAMIX ontology is written in OWL and has been designed using the Protege[1] tool. After each step, the ontology has been verified for consistency using the built-in reasoner of the Protege tool.
The translation of the meta model into an ontology proceeds as follows:

1. The meta classes are transformed to OWL classes according to TR1. It is necessary to first define the ontology classes because they need to be referenced in the following declarations.
2. Attributes of the meta classes are transformed to data type properties according to TR2 to TR4
3. The generalization relationships are transformed to subclass axioms according to TR5 and TR6
4. Associations between the meta classes are transformed to object properties according to TR7 to TR10

1. Meta Classes The meta classes in Figure 7.4 (see Section 7.2.3) are translated to OWL classes. The name of an OWL class corresponds to the name of the respective meta class it is translated from. The meta model shown in Figure 7.4 therefore results in the following declarations:

Declaration(Class(Type)), Declaration(Class(Attribute)), Declaration(Class(PrimitiveType))
Declaration(Class(Class)), Declaration(Class(Method)), Declaration(Class(Access))
Declaration(Class(Namespace)), Declaration(Class(Invocation)),
Declaration(Class(Inheritance))

[1]https://protege.stanford.edu/

Table E.1.: *Summary of the transformation rules (TR) applied to transform the FAMIX meta model to its ontology.*

Rule ID	Description	OWL Axiom in Functional-style Syntax
	Meta Model Class	
TR 1	Specify a class declaration axiom for class *name*.	*Declaration(Class (name))*
	Attributes	
TR 2	Specify a declaration axiom for a datatype property A for the attribute.	*Declaration(DatatypeProperty(A))*
TR 3	Specify the domain C of the datatype property A.	*DataPropertyDomain(A C)*
TR 4	Specify the data type range *type* of the datatype property A.	*DataPropertyRange(DatatypeProperty(P type)*
	Generalization	
TR 5	Specify a *SubClassOf* axiom for the generalization between meta model classes C and D.	*SubClassOf(C D)*
TR 6	Specify sibling concepts to be pairwise disjoint.	*DisjointClasses(*C_i C_j*)*, $i = 1,...,n;\ j = 1,...,n;\ i \neq j$
	Associations	
TR 7	Specify a declaration axiom for object property R.	*Declaration (ObjectProperty(R))*
TR 8	Specify a declaration axiom for object property S and define it to be inverse with R.	*Declaration (ObjectProperty(S))* *InverseObjectProperties (R S)*
TR 9	Specify the domain C and range D for properties R and S.	*ObjectPropertyDomain(R C)*, *ObjectPropertyRange(R D)* *ObjectPropertyDomain(S D)* *ObjectPropertyRange(S C)*
TR 10	Optional: In case the multiplicity equals 1, specify R to be a functional property.	*FunctionalProperty(R)*

2. Attributes Attributes for the meta classes Class, Attribute, and Method are defined. The name of the data type properties correspond to the names of the attributes in the meta model.

Declaration(DatatypeProperty(hasClassScope))
Declaration(DatatypeProperty(isInterface))

The isInterface attribute is only defined for instances of Class. That is why, the domain of isInterface is defined as:

DataPropertyDomain(isInterface Class)

In the FAMIX meta model, attributes with the same name are defined for **Method** and **Attribute**. As described in the previous section, attributes only have a local class scope. Therefore, hasClassScope is only defined once. As a consequence, the domain of the data type property is defined as the union of both concepts:

DataPropertyDomain(ObjectUnionOf(Method Attribute))

Both attributes store boolean values. That is why, the range of the corresponding data type properties is set to boolean (the xsd prefix references the namespace of the XML schema[2]):

DataPropertyRange(isInterface `xsd:boolean` *)*
DataPropertyRange(hasClassScope `xsd:boolean` *)*

3. Generalization As a next step, the generalization relationships are transformed. The meta classes **PrimitiveType** and **Class** are specializations of **Type**. This is expressed as follows according to the rule TR5:

SubClassOf(PrimitiveType Type)
SubClassOf(Class Type)

Individuals cannot be both a **PrimitiveType** and a **Class**. So, by definition, those concepts are pairwise disjoint (see TR6):

DisjointClasses(PrimitiveType Class)

This is not explicitly stated in the class diagram of the FAMIX meta model. However, this restriction naturally follows from the meaning - as described in [DAB$^+$11] - of each concept, e.g., a primitive type such as `int` is not a class.

4. Associations The transformation of associations is shown exemplary on the association between the concepts Namespace and Type. Both association ends are named. For both ends, an object property is defined:

Declaration(ObjectProperty(isContainedIn))
Declaration(ObjectProperty(containsType))

Next, for both properties, the domain and range constraints are defined:

[2]https://www.w3.org/XML/Schema

ObjectPropertyDomain(isContainedIn Type), ObjectPropertyRange(isContainedIn Namespace)
ObjectPropertyDomain(containsType Namespace), ObjectPropertyRange(containsType Type)

The object properties are inverse properties:

InverseObjectProperties (isContainedIn containsType)

A Namespace can contain an arbitrary number of types, whereas a type can only be contained in exactly one namespace:

FunctionalProperty(isContainedIn)

E.1.1. Ontology

The OWL representation of the ontology is integrated in the toolchain[3].

E.1.2. Class Declarations

Table E.2.: *Class declarations for OWL classes representing meta classes of the FAMIX meta model.*

Name	Axiom
Entity	*Declaration(Class(Entity))*
AnnotationInstance	*Declaration(Class(AnnotationInstance))*
AnnotationInstanceAttribute	*Declaration(Class(AnnotationInstanceAttribute))*
AnnotationTypeAttribute	*Declaration(Class(AnnotationTypeAttribute))*
SourceAnchor	*Declaration(Class(SourceAnchor))*
SourceLanguage	*Declaration(Class(SourceLanguage))*
SourcedEntity	*Declaration(Class(SourcedEntity))*
Association	*Declaration(Class(Association))*
Comment	*Declaration(Class(Comment))*
NamedEntity	*Declaration(Class(NamedEntity))*
Access	*Declaration(Class(Access))*
Inheritance	*Declaration(Class(Inheritance))*
Invocation	*Declaration(Class(Invocation))*
Reference	*Declaration(Class(Reference))*
ContainerEntity	*Declaration(Class(ContainerEntity))*
LeafEntity	*Declaration(Class(LeafEntity))*
BehaviouralEntity	*Declaration(Class(BehaviouralEntity))*
ScopingEntity	*Declaration(Class(ScopingEntity))*
Type	*Declaration(Class(Type))*

Continued on next page

[3]https://github.com/sandrellaella/owlify/blob/master/owlify-famix/ontology/famix.owl, Last access: 30 July 2019

Table E.2 – continued from previous page

Name	**Axiom**
StructuralEntity	*Declaration(Class(StructuralEntity))*
Function	*Declaration(Class(Function))*
Method	*Declaration(Class(Method))*
Namespace	*Declaration(Class(Namespace))*
Package	*Declaration(Class(Package))*
AnnotationType	*Declaration(Class(AnnotationType))*
Class	*Declaration(Class(Class))*
PrimitiveType	*Declaration(Class(PrimitiveType))*
Attribute	*Declaration(Class(Attribute))*
GlobalVariable	*Declaration(Class(GlobalVariable))*
LocalVariable	*Declaration(Class(LocalVariable))*
Parameter	*Declaration(Class(Parameter))*
AnnotationTypeAttribute	*Declaration(Class(AnnotationTypeAttribute))*
Exception	*Declaration(Class(Exception))*
CaughtException	*Declaration(Class(CaughtException))*
DeclaredException	*Declaration(Class(DeclaredException))*
ThrownException	*Declaration(Class(ThrownException))*
Enum	*Declaration(Class(Enum))*
ParameterType	*Declaration(Class(ParameterType))*
ParameterizedType	*Declaration(Class(ParameterizedType))*
EnumValue	*Declaration(Class(EnumValue))*
ParameterizableClass	*Declaration(Class(ParameterizableClass))*

E.1.3. Generalization

The results of the transformation are given in Table E.3 on the next page (page 237).

Table E.3.: *Generalization between OWL classes according to the FAMIX meta model. The expression A → B represents a generalization between the OWL classes A and B, where A is a sub-class of B.*

Related Classes	**Axiom**
AnnotationInstance → Entity	*SubClassOf(AnnotationInstance Entity)*
AnnotationInstanceAttribute → Entity	*SubClassOf(AnnotationInstanceAttribute Entity)*
SourceAnchor → Entity	*SubClassOf(SourceAnchor Entity)*
SourceLanguage → Entity	*SubClassOf(SourceLanguage Entity)*
SourcedEntity → Entity	*SubClassOf(SourcedEntity Entity)*
Association → SourcedEntity	*SubClassOf(Association SourcedEntity)*
Comment → SourcedEntity	*SubClassOf(Comment SourcedEntity)*
NamedEntity → SourcedEntity	*SubClassOf(NamedEntity SourcedEntity)*
Access → Association	*SubClassOf(Access Association)*
Inheritance → Association	*SubClassOf(Inheritance Association)*
Invocation → Association	*SubClassOf(Invocation Association)*
Reference → Association	*SubClassOf(Reference Association)*
ContainerEntity → NamedEntity	*SubClassOf(ContainerEntity NamedEntity)*
LeafEntity → ContainerEntity	*SubClassOf(LeafEntity ContainerEntity)*
BehavioralEntity → ContainerEntity	*SubClassOf(BehavioralEntity ContainerEntity)*
ScopingEntity → ContainerEntity	*SubClassOf(ScopingEntity ContainerEntity)*
Type → ContainerEntity	*SubClassOf(Type ContainerEntity)*
StructuralEntity → LeafEntity	*SubClassOf(LeafEntity ContainerEntity)*

Continued on next page

Table E.3 – continued from previous page

Related Classes	**Axiom**
Function → BehavioralEntity	*SubClassOf(Function BehavioralEntity)*
Method → BehavioralEntity	*SubClassOf(Method BehavioralEntity)*
Namespace → ScopingEntity	*SubClassOf(Namespace ScopingEntity)*
Package → ScopingEntity	*SubClassOf(Package ScopingEntity)*
AnnotationType → Type	*SubClassOf(AnnotationType Type)*
Class → Type	*SubClassOf(Class Type)*
PrimitiveType → Type	*SubClassOf(PrimitiveType Type)*
Attribute → StructuralEntity	*SubClassOf(Attribute StructuralEntity)*
GlobalVariable → StructuralEntity	*SubClassOf(GlobalVariable StructuralEntity)*
LocalVariable → StructuralEntity	*SubClassOf(LocalVariable StructuralEntity)*
Parameter → StructuralEntity	*SubClassOf(Parameter StructuralEntity)*
AnnotationTypeAttribute → Attribute	*SubClassOf(AnnotationTypeAttribute StructuralEntity)*
Exception → Entity	*SubClassOf(Exception Entity)*
CaughtException → Exception	*SubClassOf(CaughtException Exception)*
DeclaredException → Exception	*SubClassOf(DeclaredException Exception)*
ThrownException → Exception	*SubClassOf(ThrownException Exception)*
Enum → Type	*SubClassOf(Enum Type)*
ParameterType → Type	*SubClassOf(ParameterType Type)*
ParameterizedType → Type	*SubClassOf(ParameterizedType Type)*

Continued on next page

Table E.3 – continued from previous page

Related Classes	**Axiom**
EnumValue → StructuralEntity	*SubClassOf(EnumValue StructuralEntity)*
ParameterizableClass → Class	*SubClassOf(ParameterizableClass Class)*

E.1.4. Object Properties

The results of the transformation are given in Table E.4 on the next page (page 241).

Table E.4.: *Object properties corresponding to bidirectional associations between meta model classes.*

Name	**Axiom**

annotationInstance	*Declaration(ObjectProperty(hasAnnotationInstance))*
	ObjectPropertyDomain(hasAnnotationInstance Entity)
	ObjectPropertyRange(hasAnnotationInstance AnnotationInstance)

annotationType	*Declaration(ObjectProperty(hasAnnotationType))*
	ObjectPropertyDomain(hasAnnotationType AnnotationInstance)
	ObjectPropertyRange(hasAnnotationType AnnotationType)

annotatedEntity	*Declaration(ObjectProperty(annotatesEntity))*
	ObjectPropertyDomain(annotatesEntity AnnotationInstance)
	ObjectPropertyRange(annotatesEntity Entity)

attributes	*Declaration(ObjectProperty(hasAttribute))*
	ObjectPropertyDomain(hasAttribute AnnotationInstance)
	ObjectPropertyRange(hasAttribute AnnotationInstanceAttribute)

parentAnnotationInstance	*Declaration(ObjectProperty(hasParentAnnotationInstance))*
	ObjectPropertyDomain(hasParentAnnotationInstance AnnotationInstanceAttribute)
	ObjectPropertyRange(hasParentAnnotationInstance AnnotationInstance)
	InverseObjectProperties(hasParentAnnotationInstance hasAttribute)

hasAnnotationTypeAttribute	*Declaration(ObjectProperty(hasAnnotationTypeAttribute))*
	ObjectPropertyDomain(hasAnnotationTypeAttribute AnnotationInstanceAttribute)
	ObjectPropertyRange(hasAnnotationTypeAttribute AnnotationTypeAttribute)

hasElement	*Declaration(ObjectProperty(hasElement))*
	ObjectPropertyDomain(hasElement SourceAnchor)
	ObjectPropertyRange(hasElement SourcedEntity)

hasSourcedEntity	*Declaration(ObjectProperty(hasSourcedEntity))*
	ObjectPropertyDomain(hasSourcedEntity SourceLanguage)

Table E.4 – continued from previous page

Name	Axiom
	ObjectPropertyRange(hasSourcedEntity SourcedEntity)

hasSourceAnchor	*Declaration(ObjectProperty(hasSourceAnchor))*
	ObjectPropertyDomain(hasSourceAnchor SourcedEntity)
	ObjectPropertyRange(hasSourceAnchor SourceAnchor)
	InverseObjectProperties(hasSourceAnchor hasElement)

hasDeclaredSourceLanguage	*Declaration(ObjectProperty(hasDeclaredSourceLanguage))*
	ObjectPropertyDomain(hasDeclaredSourceLanguage SourcedEntity)
	ObjectPropertyRange(hasDeclaredSourceLanguage SourceLanguage)
	InverseObjectProperties(hasDeclaredSourceLanguage hasSourcedEntity)

comments	*Declaration(ObjectProperty(hasComment))*
	ObjectPropertyDomain(hasComment SourcedEntity)
	ObjectPropertyRange(hasComment Comment)

previous	*Declaration(ObjectProperty(hasPrevious))*
	ObjectPropertyDomain(hasPrevious Association)
	ObjectPropertyRange(hasPrevious Association)

next	*Declaration(ObjectProperty(hasNext))*
	ObjectPropertyDomain(hasNext Association)
	ObjectPropertyRange(hasNext Association)
	InverseObjectProperties(hasNext hasPrevious)

from	*Declaration(ObjectProperty(from))*
	ObjectPropertyDomain(from Association)
	ObjectPropertyRange(from NamedEntity)

to	*Declaration(ObjectProperty(to))*
	ObjectPropertyDomain(to Association)

Table E.4 – continued from previous page

Name	**Axiom**
	ObjectPropertyRange(to NamedEntity)

container (Comment)	*Declaration(ObjectProperty(containedInSourcedEntity))*
	ObjectPropertyDomain(containedInSourcedEntity Comment)
	ObjectPropertyRange(containedInSourcedEntity SourcedEntity)
	InverseObjectProperties(containedInSourcedEntity hasComment)

receivingInvocations	*Declaration(ObjectProperty(receivesInvocation))*
	ObjectPropertyDomain(receivesInvocation NamedEntity)
	ObjectPropertyRange(receivesInvocation Invocation)

parentPackage	*Declaration(ObjectProperty(hasParentPackage))*
	ObjectPropertyDomain(hasParentPackage NamedEntity)
	ObjectPropertyRange(hasParentPackage Package)

belongsTo	*Declaration(ObjectProperty(belongsTo))*
	ObjectPropertyDomain(belongsTo NamedEntity)
	ObjectPropertyRange(belongsTo ContainerEntity)

accessor	*Declaration(ObjectProperty(hasAccessor))*
	ObjectPropertyDomain(hasAccessor Access)
	ObjectPropertyRange(hasAccessor BehavioralEntity)

variable	*Declaration(ObjectProperty(accessesVariable))*
	ObjectPropertyDomain(accessesVariable Access)
	ObjectPropertyRange(accessesVariable StructuralEntity)

subClass	*Declaration(ObjectProperty(hasSubClass))*
	ObjectPropertyDomain(hasSubClass Inheritance)
	ObjectPropertyRange(hasSubClass Type)

superClass	*Declaration(ObjectProperty(hasSuperClass))*

Table E.4 – continued from previous page

Name	Axiom

	ObjectPropertyDomain(hasSuperClass Inheritance)
	ObjectPropertyRange(hasSuperClass Type)

candidates	*Declaration(ObjectProperty(hasCandidate))*
	ObjectPropertyDomain(hasCandidate Invocation)
	ObjectPropertyRange(hasCandidate BehavioralEntity)

sender	*Declaration(ObjectProperty(hasSender))*
	ObjectPropertyDomain(hasSender Invocation)
	ObjectPropertyRange(hasSender BehavioralEntity)

receiver	*Declaration(ObjectProperty(hasReceiver))*
	ObjectPropertyDomain(hasReceiver Invocation)
	ObjectPropertyRange(hasReceiver NamedEntity)

target	*Declaration(ObjectProperty(hasTarget))*
	ObjectPropertyDomain(hasTarget Reference)
	ObjectPropertyRange(hasTarget ContainerEntity)
	FunctionalProperty(hasTarget)

source	*Declaration(ObjectProperty(hasSource))*
	ObjectPropertyDomain(hasSource Reference)
	ObjectPropertyRange(hasSource ContainerEntity)
	FunctionalProperty(hasSource)

incomingReferences	*Declaration(ObjectProperty(hasIncomingReference))*
	ObjectPropertyDomain(hasIncomingReference ContainerEntity)
	ObjectPropertyRange(hasIncomingReference ContainerEntity)
	InverseObjectProperties(hasIncomingReference hasTarget)

types	*Declaration(ObjectProperty(containsType))*

Table E.4 – continued from previous page

Name	**Axiom**

	ObjectPropertyDomain(containsType ContainerEntity)
	ObjectPropertyRange(containsType Type)

outgoingReferences	*Declaration(ObjectProperty(hasOutgoingReference))*
	ObjectPropertyDomain(hasOutgoingReference ContainerEntity)
	ObjectPropertyRange(hasOutgoingReference Reference)
	InverseObjectProperties(hasOutgoingReference hasSource)

definedAnnotationTypes	*Declaration(ObjectProperty(definesAnnotationType))*
	ObjectPropertyDomain(definesAnnotationType ContainerEntity)
	ObjectPropertyRange(definesAnnotationType AnnotationType)

incomingInvocations	*Declaration(ObjectProperty(hasIncomingInvocation))*
	ObjectPropertyDomain(hasIncomingInvocation BehavioralEntity)
	ObjectPropertyRange(hasIncomingInvocation Invocation)
	InverseObjectProperties(hasIncomingInvocation hasCandidate)

localVariables	*Declaration(ObjectProperty(hasLocalVariable))*
	ObjectPropertyDomain(hasLocalVariable BehavioralEntity)
	ObjectPropertyRange(hasLocalVariable LocalVariable)

accesses	*Declaration(ObjectProperty(hasAccess))*
	ObjectPropertyDomain(hasAccess BehavioralEntity)
	ObjectPropertyRange(hasAccess Access)
	InverseObjectProperties(hasAccessor hasAccess)

outgoingInvocations	*Declaration(ObjectProperty(hasOutgoingInvocation))*
	ObjectPropertyDomain(hasOutgoingInvocation BehavioralEntity)
	ObjectPropertyRange(hasOutgoingInvocation Invocation)
	InverseObjectProperties(hasOutgoingInvocation hasSender)

Table E.4 – continued from previous page

Name	**Axiom**

declaredType	*Declaration(ObjectProperty(hasDeclaredType))*
	 ObjectPropertyDomain(hasDeclaredType ObjectUnionOf(BehavioralEntity StructuralEntity))
	 ObjectPropertyRange(hasDeclaredType Type)

parameters	*Declaration(ObjectProperty(hasParameter))*
	 ObjectPropertyDomain(hasParameter BehavioralEntity)
	 ObjectPropertyRange(hasParameter Parameter)

childScopes	*Declaration(ObjectProperty(hasChildScope))*
	 ObjectPropertyDomain(hasChildScope ScopingEntity)
	 ObjectPropertyRange(hasChildScope ScopingEntity)

functions	*Declaration(ObjectProperty(hasFunction))*
	 ObjectPropertyDomain(hasFunction ScopingEntity)
	 ObjectPropertyRange(hasFunction Function)

globalVariables	*Declaration(ObjectProperty(hasGlobalVariable))*
	 ObjectPropertyDomain(hasGlobalVariable ScopingEntity)
	 ObjectPropertyRange(hasGlobalVariable GlobalVariable)

parentScope	*Declaration(ObjectProperty(hasParentScope))*
	 ObjectPropertyDomain(hasParentScope ObjectUnion(ScopingEntity Function GlobalVariable))
	 ObjectPropertyRange(hasParentScope ScopingEntity)

methods	*Declaration(ObjectProperty(hasMethod))*
	 ObjectPropertyDomain(hasMethod Type)
	 ObjectPropertyRange(hasMethod Method)

container (Type)	*Declaration(ObjectProperty(isContainedIn))*

Table E.4 – continued from previous page

Name	**Axiom**

	ObjectPropertyDomain(isContainedIn Type)
	ObjectPropertyRange(isContainedIn ContainerEntity)
	InverseObjectProperties(isContainedIn containsType)

attributes	*Declaration(ObjectProperty(hasAttribute))*
	ObjectPropertyDomain(hasAttribute Type)
	ObjectPropertyRange(hasAttribute Attribute)

behavioursWithDeclaredType	*Declaration(ObjectProperty(isDeclaredTypeOf))*
	ObjectPropertyDomain(isDeclaredTypeOf Type)
	ObjectPropertyRange(isDeclaredTypeOf ObjectUnionOf(BehavioralEntity Struc-turalEntity))
	InverseObjectProperties(isDeclaredTypeOf hasDeclaredType)

inComingAccesses	*Declaration(ObjectProperty(hasIncomingAccess))*
	ObjectPropertyDomain(hasIncomingAccess StructuralEntity)
	ObjectPropertyRange(hasIncomingAccess Access)
	InverseObjectProperties(hasIncomingAccess hasVariable)

parentModule	*Declaration(ObjectProperty(parentModule))*
	ObjectPropertyDomain(parentModule ObjectUnionOf(Function GlobalVariable))
	ObjectPropertyRange(parentModule Module)

caughtExceptions	*Declaration(ObjectProperty(caughtException))*
	ObjectPropertyDomain(caughtException Method)
	ObjectPropertyRange(caughtException CaughtException)

thrownExceptions	*Declaration(ObjectProperty(thrownException))*
	ObjectPropertyDomain(thrownException Method)
	ObjectPropertyRange(thrownException ThrownException)

Table E.4 – continued from previous page

Name	**Axiom**

declaredExceptions	*Declaration(ObjectProperty(declaredException))*
	ObjectPropertyDomain(declaredException Method)
	ObjectPropertyRange(declaredException ThrownException)

parentType	*Declaration(ObjectProperty(hasParentType))*
	ObjectPropertyDomain(hasParentType ObjectUnionOf(Method StructuralEntity))
	ObjectPropertyRange(hasParentType Type)
	FunctionalProperty(hasParentType)
	InverseObjectProperty(hasParentType hasMethod)
	InverseObjectProperty(hasParentType hasAttribute)

childNamedEntities	*Declaration(ObjectProperty(hasChildNamedEntity))*
	ObjectPropertyDomain(hasChildNamedEntity Package)
	ObjectPropertyRange(hasChildNamedEntity NamedEntity)
	InverseObjectProperty(hasParentType hasParentPackage)

instances	*Declaration(ObjectProperty(hasInstance))*
	ObjectPropertyDomain(hasInstance AnnotationType)
	ObjectPropertyRange(hasInstance AnnotationType)
	InverseObjectProperty(hasInstance hasAnnotationType)

parentBehavioralEntity	*Declaration(ObjectProperty(parentBehavioralEntity))*
	ObjectPropertyDomain(parentBehavioralEntity ObjectUnionOf(LocalVariable Parameter))
	ObjectPropertyRange(parentBehavioralEntity BehavioralEntity)
	FunctionalProperty(parentBehavioralEntity)
	InverseObjectProperty(parentBehavioralEntity hasLocalVariable)

annotationAttributeInstances	*Declaration(ObjectProperty(hasAnnotationAttributeInstance))*
	ObjectPropertyDomain(hasAnnotationAttributeInstance AnnotationTypeAttribute)

Table E.4 – continued from previous page

Name	**Axiom**

	ObjectPropertyRange(hasAnnotationAttributeInstance AnnotationInstanceAttribute)
	FunctionalProperty(hasAnnotationAttributeInstance)
	InverseObjectProperty(hasAnnotationAttributeInstance hasAnnotationTypeAttribute)

exceptionClass	*Declaration(ObjectProperty(hasExceptionClass))*
	ObjectPropertyDomain(hasExceptionClass Exception)
	ObjectPropertyRange(hasExceptionClass Class)
	FunctionalProperty(hasExceptionClass)

definingMethod	*Declaration(ObjectProperty(definingMethod))*
	ObjectPropertyDomain(definingMethod ObjectUnionOf(CaughtException DeclaredException DeclaredException))
	ObjectPropertyRange(definingMethod Method)
	FunctionalProperty(definingMethod)
	InverseObjectProperty(definingMethod caughtException)

values	*Declaration(ObjectProperty(hasEnumValue))*
	ObjectPropertyDomain(hasEnumValue Enum))
	ObjectPropertyRange(hasEnumValue EnumValue)

arguments	*Declaration(ObjectProperty(hasArgument))*
	ObjectPropertyDomain(hasArgument ParameterizedType))
	ObjectPropertyRange(hasArgument Type)

parameterizableClass	*Declaration(ObjectProperty(parameterizableClass))*
	ObjectPropertyDomain(parameterizableClass ParameterizedType))
	ObjectPropertyRange(parameterizableClass ParameterizableClass)

parentEnum	*Declaration(ObjectProperty(parentEnum))*
	ObjectPropertyDomain(parentEnum EnumValue))
	ObjectPropertyRange(parentEnum Enum)

Table E.4 – continued from previous page

Name	**Axiom**
	InverseObjectProperties(parentEnum hasEnumValue)

parameters (ParameterizableClass)	*Declaration(ObjectProperty(hasParameterType))*
	ObjectPropertyDomain(hasParameterType ParameterizableClass))
	ObjectPropertyRange(hasParameterType ParameterType)

E.1.5. Data Properties

The results of the transformation are given in Table E.5 on the next page (page 252).

Table E.5.: *Data properties corresponding to attributes of meta model classes.*

Name	**Axiom**
value	*Declaration(DatatypeProperty(hasValue))*
	DataPropertyDomain(hasValue AnnotationInstanceAttribute)
	DataPropertyRange(hasValue `xsd:String` *)*
	FunctionalProperty(hasValue)
content	*Declaration(DatatypeProperty(hasContent))*
	DataPropertyDomain(hasContent Comment)
	DataPropertyRange(hasContent `xsd:String` *)*
	FunctionalProperty(hasContent)
isPublic	*Declaration(DatatypeProperty(isPublic))*
	DataPropertyDomain(isPublic NamedEntity)
	DataPropertyRange(isPublic `xsd:Boolean` *)*
	FunctionalProperty(isPublic)
isPrivate	*Declaration(DatatypeProperty(isPrivate))*
	DataPropertyDomain(isPrivate NamedEntity)
	DataPropertyRange(isPrivate `xsd:Boolean` *)*
	FunctionalProperty(isPrivate)
isFinal	*Declaration(DatatypeProperty(isFinal))*
	DataPropertyDomain(isFinal NamedEntity)
	DataPropertyRange(isFinal `xsd:Boolean` *)*
	FunctionalProperty(isFinal)
isProtected	*Declaration(DatatypeProperty(isProtected))*
	DataPropertyDomain(isProtected NamedEntity)
	DataPropertyRange(isProtected `xsd:Boolean` *)*
	FunctionalProperty(isProtected)
nameLength	*Declaration(DatatypeProperty(hasNameLength))*

Continued on next page

Table E.5 – continued from previous page

Name	Axiom
	DataPropertyDomain(hasNameLength NamedEntity)
	DataPropertyRange(hasNameLength `xsd:unsignedLong` *)*
	FunctionalProperty(hasNameLength)
isAbstract	*Declaration(DatatypeProperty(isAbstract))*
	DataPropertyDomain(isAbstract ObjectUnionOf(NamedEntity Type)
	DataPropertyRange(isAbstract `xsd:Boolean` *)*
	FunctionalProperty(isAbstract)
modifiers	*Declaration(DatatypeProperty(hasModifier))*
	DataPropertyDomain(hasModifier NamedEntity)
	DataPropertyRange(hasModifier `xsd:String` *)*
isStub	*Declaration(DatatypeProperty(isStub))*
	DataPropertyDomain(isStub NamedEntity)
	DataPropertyRange(isStub `xsd:Boolean` *)*
	FunctionalProperty(isStub)
isRead	*Declaration(DatatypeProperty(isRead))*
	DataPropertyDomain(isRead Access)
	DataPropertyRange(isRead `xsd:Boolean` *)*
	FunctionalProperty(isRead)
isWrite	*Declaration(DatatypeProperty(isWrite))*
	DataPropertyDomain(isWrite Access)
	DataPropertyRange(isWrite `xsd:Boolean` *)*
	FunctionalProperty(isWrite)
signature	*Declaration(DatatypeProperty(hasSignature))*
	DataPropertyDomain(hasSignature ObjectUnionOf(Invocation BehavioralEntity))
	DataPropertyRange(hasSignature `xsd:String` *)*

Continued on next page

Table E.5 – continued from previous page

Name	Axiom
	FunctionalProperty(hasSignature)
numberOfMessageSend	*Declaration(DatatypeProperty(numberOfMessageSend))*
	DataPropertyDomain(numberOfMessageSend BehavioralEntity)
	DataPropertyRange(numberOfMessageSend `xsd:unsignedLong` *)*
	FunctionalProperty(numberOfMessageSend)
numberOfStatements	*Declaration(DatatypeProperty(numberOfStatements))*
	DataPropertyDomain(numberOfStatements BehavioralEntity)
	DataPropertyRange(numberOfStatements `xsd:unsignedLong` *)*
	FunctionalProperty(numberOfStatements)
numberOfAccesses	*Declaration(DatatypeProperty(numberOfAccesses))*
	DataPropertyDomain(numberOfAccesses BehavioralEntity)
	DataPropertyRange(numberOfStatements `xsd:unsignedLong` *)*
	FunctionalProperty(numberOfAccesses)
numberOfLinesOfCode	*Declaration(DatatypeProperty(numberOfLinesOfCode))*
	DataPropertyDomain(numberOfLinesOfCode BehavioralEntity)
	DataPropertyRange(numberOfLinesOfCode `xsd:unsignedLong` *)*
	FunctionalProperty(numberOfLinesOfCode)
isSetter	*Declaration(DatatypeProperty(isSetter))*
	DataPropertyDomain(isSetter Method)
	DataPropertyRange(isSetter `xsd:Boolean` *)*
	FunctionalProperty(isSetter)
hasClassScope	*Declaration(DatatypeProperty(hasClassScope))*
	DataPropertyDomain(hasClassScope ObjectUnionOf(Method Attribute))
	DataPropertyRange(hasClassScope `xsd:Boolean` *)*
	FunctionalProperty(hasClassScope)
isGetter	*Declaration(DatatypeProperty(isGetter))*

Continued on next page

Table E.5 – continued from previous page

Name	Axiom
	DataPropertyDomain(isGetter Method)
	DataPropertyRange(isGetter `xsd:Boolean` *)*
	FunctionalProperty(isGetter)
timeStamp	Declaration(DatatypeProperty(hasTimeStamp))
	DataPropertyDomain(hasTimeStamp Method)
	DataPropertyRange(hasTimeStamp `xsd:String` *)*
	FunctionalProperty(hasTimeStamp)
isConstant	*Declaration(DatatypeProperty(isConstant))*
	DataPropertyDomain(isConstant Method)
	DataPropertyRange(isConstant `xsd:Boolean` *)*
	FunctionalProperty(isConstant)
kind	*Declaration(DatatypeProperty(isKind))*
	DataPropertyDomain(isKind Method)
	DataPropertyRange(isKind `xsd:String` *)*
	FunctionalProperty(isKind)
isOverriden	*Declaration(DatatypeProperty(isOverriden))*
	DataPropertyDomain(isOverriden Method)
	DataPropertyRange(isOverriden `xsd:Boolean` *)*
	FunctionalProperty(isOverriden)
isOverriding	*Declaration(DatatypeProperty(isOverriding))*
	DataPropertyDomain(isOverriding Method)
	DataPropertyRange(isOverriding `xsd:Boolean` *)*
	FunctionalProperty(isOverriding)
isInternalImplementation	*Declaration(DatatypeProperty(isInternalImplementation))*
	DataPropertyDomain(isInternalImplementation Method)

Continued on next page

Table E.5 – continued from previous page

Name	Axiom
	DataPropertyRange(isInternalImplementation `xsd:Boolean` *)*
	FunctionalProperty(isInternalImplementation)
isInterface	*Declaration(DatatypeProperty(isInterface))*
	DataPropertyDomain(isInterface Class)
	DataPropertyRange(isInterface `xsd:Boolean` *)*
	FunctionalProperty(isInterface)

F. Populating Conformance Check Results

The steps are conducted in three separate phases, the *preparation phase*, the *creation phase* and the *unification phase*. In the preparation phase, the proof trees are calculated that are processed and stored in the subsequent phases. In the creation phase, the individuals are created that describe the conformance check. This includes creating individuals of the concepts ConformanceCheck, ArchitectureViolation, and ArchitectureRule. In the unification phase, all individuals are connected with each other according to the conformance check ontology, e.g., individuals of code elements are connected with individuals of the ArchitectureViolation concept. After executing the phases, the results are finally stored in the knowledge base.

Algorithm 1 depicts the main procedure for storing the architecture conformance checking results. It validates a selected set of architecture rules and processes the resulting proof trees. The algorithm expects the date on which the conformance check has been conducted, the rules that have been validated, and the knowledge base where the results should be unified with the ontology-based source code facts, the implemented architecture and the architecture ontology. According to the two phases, the creation and the unification phase, the process is separated into two procedures (Algorithm 2 and Algorithm 3). In the following, the phases and steps are described in more detail.

Predefined functions and procedures. The following algorithms (Algorithm 1 to 3) make use of several functions and procedures in order to create and retrieve individuals for which no detailed algorithms are provided. Therefore, they are briefly introduced and described in the following:

- Create*Individual: A function that creates and returns an individual for a conformance check, rule, or violation concept as defined in the conformance check ontology. These functions take no arguments.
- addLiteral(i, p, l): A procedure that connects an individual i with a literal l via the data type property p.
- addObjectProperty(i_1, p, i_2): A procedure that connects an individual i_1 with another individual i_2 via the object property p.
- validate($rule$): Performs conformance checking on a rule r and returns the set of proof trees PT_r for this rule. PT_r is empty when on violations are found for r.
- getProofTreeOfViolationIndividual(i): A function that returns the proof tree that is associated with the individual i.
- getElement(uri,KB): A function that retrieves the individual specified by its URI uri from the knowledge base KB.

- Further, the following functions on architecture rules R are defined:
  - $cnl : R \rightarrow String$: A function that returns the *ArchCNL* representation of r.
  - $type : R \rightarrow \{is-a, must, can-only, only-can...\}$: A function that returns the type of rule r.
  - $id : R \rightarrow \mathbb{N}$: A function that returns the ID of the rule.

1) Preparation. First, a conformance check individual is created that represents the actual conformance check (Algorithm 1, line 3). It is associated with the date of execution via a data type property assertion (Algorithm 1, line 4). The procedure VALIDATE calls the actual conformance checking process and returns a set of proof trees representing the set of violations of this rule (Algorithm 1, line 6). The set of proof trees is empty when no violations have been detected.

In case violations have been found, the corresponding individuals are created to store the violations (line 7, implemented by Algorithm 2) and the results are unified in the knowledge base (line 8, implemented by Algorithm 3). In a final step, the results are stored in the knowledge base (line 10).

Algorithm 1 Main procedure separated in a creation phase (CREATE) and a unification phase (UNIFY) for storing the conformance checking results.

```
1: procedure storeConformanceCheckingResults(date, version, name, $ACL$, $IA$)
2:     $ACC_{KB} \leftarrow ACL \cup IA$     ▷ knowledge base combining the architecture ontology and the implemented architecture
3:     conformanceCheckIndividual ← createIndividual(ConformanceCheck, name)
4:     addLiteral(executedOn, conformanceCheckIndividual, date)
5:     for all $r \in R$ do
6:         $PT_r \leftarrow$ validate($r$)
7:         violationIndividuals ← create($r$, $PT_r$) ▷ creation phase, if $PT_r = \emptyset$, then $violationsIndividuals = \emptyset$
8:         unify(conformanceCheckIndividual, version, ruleIndividual, violationIndividuals, $ACC_{KB}$)     ▷ unification phase
9:     end for
10:    storeToKnowledgeBase(KB)
11: end procedure
```

2) Creation phase. Algorithm 2 shows the necessary steps of the creation phase. The procedure described in this algorithm is responsible for creating the individuals for the current rule and the violations extracted from the proof trees. The procedure CREATE expects the architecture rule and the proof trees that describe the corresponding violations of the rule. For each architecture rule that has been validated, an individual is created (Algorithm 2, line 3). An architecture rule is associated with its *ArchCNL* representation (Algorithm 2, line 4), the rule type (Algorithm 2, line 5), and an ID (Algorithm 2, line 6). For each proof tree, an individual of the class ARCHITECTUREVIOLATION is created (Algorithm 2, line 8). It is associated with the text representing the proof tree, e.g., as shown in Listing 7.1, by the data type property isProofedBy (Algorithm 2, line 9).

3) Unification phase. After creating the conformance check individuals, the rule and violation individuals in the previous phases, they can be connected with each other and unified with existing facts from the knowledge base. Algorithm 3 depicts the necessary steps. First,

Algorithm 2 Create an individual for an architecture rules and create individuals for its violations - if there are any - by traversing proof trees calculated for this rule.

```
1: function CREATE($r$, $PT_r$)
2:     violationIndividuals ← $\emptyset$
3:     ruleIndividual ← CREATERULEINDIVIDUAL()
4:     ADDLITERAL(hasCNLRepresentation, ruleIndividual, $cnl(r)$)
5:     ADDLITERAL(hasRuleType, ruleIndividual, $type(r)$)
6:     ADDLITERAL(hasId, ruleIndividual, $id(r)$)
7:     for all $pt \in PT_r$ do
8:         violationIndividual ← CREATEVIOLATIONINDIVIDUAL()
9:         ADDLITERAL(isProofedBy, violationIndividual, $pt$)
10:        violationIndividuals ← violationIndividuals ∪ violationIndividual
11:    end for
12:    return ruleIndividual, violationIndividuals
13: end function
```

the conformance check individual is connected with the rule individual representing the rule that has been validated in this conformance check (Algorithm 3, line 2). Based on the specified version, e.g., given by the analyzed Git repository, the corresponding individual is retrieved from the knowledge base (Algorithm 3, line 3). The individual representing this version is connected with the conformance check individual created earlier in the creation phase (Algorithm 3, line 4). Next, the violations of the current rules are processed and enriched with additional information about the violation (Algorithm 3, line 5). In lines 7 to 9, the subject, predicate, and object values are set for the violation individual. The proof tree has been retrieved previously (Algorithm 3, line 6). Since the entire URI of these individuals of the implemented architecture and the architecture ontology is given in the proof tree, they can be directly retrieved from the knowledge base. These individuals are then connected with the individual representing the violation at hand via the relation hasSubject (Algorithm 3, line 10), hasPredicate (Algorithm 3, line 11), and hasObject (Algorithm 3, line 12), respectively. The conformance check individual is connected with the violations that have been detected in this check (Algorithm 3, line 13). Finally, architecture rule individuals and architecture violation individuals are related with each other via the relations violates and hasViolation (Algorithm 3, line 14 and 15).

F.1. Retrieving Conformance Checking Results

Having stored the information about the conformance check in the knowledge base, the results can be queried from the database in order to access former architecture conformance checking results. The required information is retrieved by performing SPARQL queries. The following use cases/queries are of special interest:

1. Get conformance check and the detected violations from a specific version.
2. Get all violations of a selected rule in a specific version.
3. Get all violations and all involved code entities of a selected rule for all versions.
4. Get all rules that a selected code entity violates.

For each use case, a corresponding SPARQL query is provided that guides software architects and developers to retrieve information about specific conformance checks. In the following, the

Algorithm 3 Unify individuals into knowledge base.

```
 1: procedure UNIFY(ConformanceCheckIndividual, version, ruleIndividual, violationIndividuals, KB)
 2:     ADDOBJECTPROPERTY(VALIDATES, CONFORMANCECHECKINDIVIDUAL, RULEINDIVIDUAL)
 3:     VERSIONINDIVIDUAL ← GETVERSION(VERSION, KB)        ▷ get version individual from git ontology
 4:     ADDOBJECTPROPERTY(VALIDATESVERSION, CONFORMANCECHECKINDIVIDUAL, VERSIONINDIVIDUAL)
 5:     for all violationIndividual ∈ violationIndividuals do
 6:         pt ← GETPROOFTREEOFVIOLATIONINDIVIDUAL(violationIndividual)
 7:         subject ← GETELEMENT(subject(pt), KB)
 8:         predicate ← GETELEMENT(predicate(pt), KB)
 9:         object ← GETELEMENT(object(pt), KB)
10:         ADDOBJECTPROPERTY(hasSubject, violation, subject)
11:         ADDLITERAL(hasPredicate, violation, predicate)
12:         ADDOBJECTPROPERTY(hasObject, violation, object)
13:         ADDOBJECTPROPERTY(hasDetectedViolation, conformanceCheckIndividual, violation)
14:         ADDOBJECTPROPERTY(violates, violationIndividual, rule)
15:         ADDOBJECTPROPERTY(hasViolation, rule, violationIndividual)
16:     end for
17: end procedure
```

SPARQL queries for each use case will be explained. A SPARQL query consists of a `SELECT` and a `WHERE` clause (see Listing F.1). The `SELECT` statement lists variables that appear in the query results and that are bound to concrete values. In the example depicted in Listing F.1, the variables `?x`, `?y`, and `?z` are defined. The `WHERE` clause contains statements that are matched against the data graph. Concepts and relations from a specific ontology need to be referenced with the corresponding namespace. The namespace is abbreviated with a prefix that is a variable that stores the URI of the namespace. The prefixes `conformance`, `famix`, and `git` are prefix variables representing the corresponding namespaces of the conformance check, FAMIX, and Git ontology (see Table 7.1). The prefixes `rdf` and `rdfs` correspond to the namespace that define the classes and properties of RDF and RDFS, respectively. Since the prefix definitions are the same for all exemplary queries, the prefix definitions are left out in the SPARQL queries in the following examples in order to keep the queries clear.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX famix: <http://arch-ont.org/ontologies/famix.owl#>
PREFIX git: <http://arch-ont.org/ontologies/git.owl#>
PREFIX conformance: <http://arch-ont.org/ontologies/acc.owl#>
SELECT ?x ?y ?z
WHERE {
...
}
```

Listing F.1: *Definition of prefixes used for referring to ontology namespaces.*

F.1.1. 1. Get conformance check and the detected violations from a specific version

Listing F.2 depicts the corresponding query. It defines the variables `?check` and `?violation` that will represent the individuals of a conformance check and violations in the query result, respectively. The version - represented by the variable `?version` - is selected by its version number (42). The violations are then selected by specifying the relationship hasDetected between a conformance check and a violation as a graph matching pattern.

```
SELECT ?check ?violation
WHERE {
   ?conformanceCheck rdf:type conformance:ConformanceCheck.
   ?violation rdf:type conformance:ArchitectureViolation.
   ?version rdf:type git:Version.
   ?conformanceCheck conformance:validatesVersion ?version.
   ?version git:hasVersionNumber '42'^^xsd:integer.
   ?conformanceCheck conformance:hasDetected ?violation.
}
```

Listing F.2: *SPARQL query for retrieving the conformance check from a specific version*

F.1.2. 2. Get all violations of a selected rule in a specific version

Listing F.3 defines the corresponding SPARQL query. Similar to the previous query, the version validated by a conformance check is selected by its version number. The rule of interest is selected by its *ArchCNL* representation using the data type property hasCNLRepresentation of the conformance check ontology. In this example, all violations of the rule `Every LogicType must use a DBType.` are queried.

```
SELECT ?violation
WHERE {
   ?conformanceCheck rdf:type conformance:ConformanceCheck.
   ?conformanceCheck conformance:validatesVersion ?version.
   ?rule rdf:type conformance:ArchitectureRule.
   ?version git:hasVersionNumber '42'^^xsd:integer.
   ?conformanceCheck conformance:hasDetected ?violation.
   ?violation conformance:violates ?rule.
   ?rule conformance:hasCNLRepresentation 'Every LogicType must use a DBType
      .'^^xsd:string. }
```

Listing F.3: *SPARQL query for retrieving all violations of a selected rule from a specific version*

F.1.3. 3. Get all violations and all involved code entities of a selected rule for all versions

In the previous queries no information about the involved code entities was given that caused the violation. Only the individuals of the concept ArchitectureViolation were listed as a result. In contrast, the query in Listing F.4 retrieves all code entities that violate a selected rule. Since the conformance check ontology connects the concept NamedEntity from the FAMIX ontology with the concept ArchitectureViolation from the conformance check ontology, those entities can be easily retrieved via the properties hasSubject and hasObject. A difference to the queries before is that this query does not specify a version, since the query aims to list all violations caused by an entity across all versions of a code base.

The query will retrieve all individuals of the concept NamedEntity. This means, individuals of its subclasses, i.e. class, method, namespace, field, are listed (in case they are involved in a violation). In order to restrict the query to individuals of the concept Class, the query needs to be changed correspondingly to `?entity rdf:type famix:Class` (instead of `?entity rdf:type famix:NamedEntity`).

```
SELECT ?violation ?version ?entitySubject ?entityObject
WHERE {
   ?conformanceCheck rdf:type conformance:ConformanceCheck.
   ?conformanceCheck conformance:validatesVersion ?version.
   ?conformanceCheck conformance:validates ?rule.
   ?rule conformance:hasCNLRepresentation 'Every LogicType must use a DBType
       .'^^xsd:string.
   ?check conformance:hasDetected ?violation.
   ?entitySubject rdf:type famix:NamedEntity.
   ?entityObject rdf:type famix:NamedEntity.
   ?violation conformance:hasSubject ?entitySubject.
   ?violation conformance:hasObject ?entityObject. }
```

Listing F.4: *SPARQL query for retrieving all violations and all involved code entities of a selected rule for all versions*

F.1.4. 4. Get all rules that a selected code entity violates (across all versions)

Whereas the previous query has considered all code entities, the query in Listing F.5 selects an entity of interest by its name (`AClass`). This query then lists only the violations that are caused by this specific code entity. Note that no specific type of a code entity, e.g. class or method, is specified but only the more general concept NamedEntity. That is why, all subclasses of NamedEntity are considered. As explained in the previous section, the query needs to be changed correspondingly if individuals of specific sub-classes of NamedEntity are of interest.

```
SELECT ?rule ?version
WHERE {
   ?conformanceCheck conformance:validatesVersion ?version.
   ?violation rdf:type conformance:ArchitectureViolation.
   ?rule rdf:type conformance:ArchitectureRule.
   ?conformanceCheck conformance:hasDetected ?violation.
   ?violation conformance:violates ?rule
   ?entitySubject rdf:type famix:NamedEntity.
   ?entitySubject main:hasName 'AClass'.
   ?entityObject rdf:type famix:NamedEntity.
   ?violation conformance:hasSubject ?entitySubject.
   ?violation conformance:hasObject ?entityObject. }
```

Listing F.5: *SPARQL query for retrieving all rules that a selected code entity violates across all versions*

Bibliography

[ABO$^+$17] Nour Ali, Sean Baker, Ross O'Crowley, Sebastian Herold, and Jim Buckley. Architecture consistency: State of the practice, challenges and requirements. *Empirical Software Engineering*, pages 224–258, May 2017. (Cited on pages 23 and 78.)

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 187–197, New York, 2002. ACM. (Cited on page 87.)

[adr] Documenting architecture decisions. `http://thinkrelevance.com/blog/2011/11/15/documenting-architecture-decisions`. (Cited on pages 36, 98, and 146.)

[AGK06] Colin Atkinson, Matthias Gutheil, and Kilian Kiko. On the relationship of ontologies and models. In *2nd International Workshop on Meta-Modelling*, volume 96, pages 47–60. Ges. für Informatik, October 2006. (Cited on page 42.)

[AJLN08] Navid Ahmadi, Mehdi Jazayeri, Francesco Lelli, and Sasa Nesic. A survey of social software engineering. In *23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*, pages 1–12, September 2008. (Cited on page 21.)

[All97] Robert John Allen. *A Formal Approach to Software Architecture*. PhD thesis, Pittsburgh, PA, USA, 1997. (Cited on pages 37 and 88.)

[AMK16] Pablo Oliveira Antonino, Andreas Morgenstern, and Thomas Kuhn. Embedded-software architects: It's not only about the software. *IEEE Software*, pages 56–62, November 2016. (Cited on page 213.)

[arca] arc42-template. `https://github.com/arc42/arc42-template/tree/master/EN/asciidoc`. Last access 1.11.2019. (Cited on pages 98 and 146.)

[arcb] Archunit. `https://github.com/TNG/ArchUnit/blob/master/archunit-example/example-junit5/src/test/java/com/tngtech/archunit/exampletest/junit5/MethodReturnTypeTest.java`. Last access 31.08.2019. (Cited on page 225.)

[arcc] Archunit. `https://github.com/TNG/ArchUnit/blob/master/archunit-example/example-junit5/src/test/java/com/tngtech/archunit/exampletest/junit5/ControllerRulesTest.java`. Last access 31.08.2019. (Cited on page 227.)

[arcd] Archunit. https://github.com/TNG/ArchUnit/blob/master/archunit-example/example-junit5/src/test/java/com/tngtech/archunit/exampletest/junit5/SessionBeanRulesTest.java. Last access 31.08.2019. (Cited on page 227.)

[ARE96] Amer Al-Rawas and Steve Easterbrook. Communication problems in requirements engineering: A field study. In *Proceedings of the First Westminster Conference on Professional Awareness in Software Engineering*, pages 47–60, 1996. (Cited on page 21.)

[asc] Asciidoc home page. http://asciidoc.org/. Last access 1.11.2019. (Cited on pages 146 and 172.)

[Atl] AtlasTi. Atlasti - qualitative data analysis. https://atlasti.com/. Last access 31.10.2019. (Cited on page 61.)

[AUT] AUTOSAR. Autosar - enabling innovations. https://www.autosar.org/. Last access 31.10.2019. (Cited on page 65.)

[Bab09] M. A. Babar. An exploratory study of architectural practices and challenges in using agile software development approaches. In *Joint Working IEEE/IFIP Conference on Software Architecture European Conference on Software Architecture*, pages 81–90, Sept 2009. (Cited on page 21.)

[BCG05] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, pages 70 – 118, October 2005. (Cited on pages 126 and 127.)

[BCK12] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* Addison-Wesley Professional, Boston, 3rd edition, 2012. (Cited on pages 21, 22, 29, 30, 31, 39, and 192.)

[BCM$^+$03] Franz Baader, Diego Calvanese, Deborah McGuinness, Peter Patel-Schneider, and Daniele Nardi. *The description logic handbook: Theory, implementation and applications.* Cambridge university press, 2003. (Cited on pages 42, 43, and 48.)

[BCT07] Raffaella Bernardi, Diego Calvanese, and Camilo Thorne. Lite natural language. In *Proceedings of the 7th International Workshop on Computational Semantics*, 2007. (Cited on page 56.)

[Bec16] Petra Becker-Pechau. *Die stilbasierte Architekturprüfung: ein Ansatz zur Prüfung implementierter Softwarearchitekturen auf Architekturstil-Konformanz.* PhD thesis, University of Hamburg, 2016. (Cited on pages 82 and 83.)

[BEJV96] Pam Binns, Matt Englehart, Mike Jackson, and Steve Vestal. Domain-specific software architectures for guidance, navigation and control. *International Journal of Software Engineering and Knowledge Engineering*, pages 201–227, November 1996. (Cited on page 37.)

[Ber08] Brian Berenbach. The other skills of the software architect. In *Proceedings of the First International Workshop on Leadership and Management in Software Architecture*, pages 7–12, January 2008. (Cited on pages 212 and 213.)

[BG04] Dan Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, World Wide Web Consortium, February 2004. (Cited on page 49.)

[BGM05] Janez Brank, Marko Grobelnik, and Dunja Mladenić. A survey of ontology evaluation techniques. In *Proceedings of 8th International multi-conf. Information Society*, pages 166–169, January 2005. (Cited on page 147.)

[BH10] Frank Buschmann and Kevlin Henney. Five considerations for software architecture, part 1. *IEEE Software*, pages 63–65, May 2010. (Cited on page 212.)

[BHS05] Franz Baader, Ian Horrocks, and Ulrike Sattler. *Description Logics as Ontology Languages for the Semantic Web.* Springer Berlin Heidelberg, 2005. (Cited on page 42.)

[BL10] David Binkley and Dawn Lawrie. Information retrieval applications in software maintenance and evolution. *Encyclopedia of software engineering*, pages 454–463, 2010. (Cited on page 191.)

[BMR$^+$96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* Wiley, West Sussex, England, 1st edition, 1996. (Cited on pages 39, 84, 153, 176, 179, and 223.)

[Bos04] Jan Bosch. Software architecture: The next step. In *European Workshop on Software Architecture*, pages 194–199. Springer, 2004. (Cited on page 35.)

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement.* Springer-Verlag, Berlin, Heidelberg, 2001. (Cited on page 37.)

[BSD16] Ricardo Britto, Darja Smite, and Lars-Ola Damm. Software architects in large-scale distributed projects: An ericsson case study. *IEEE Software*, pages 48–55, November 2016. (Cited on pages 212 and 213.)

[Bus09] Frank Buschmann. Introducing the pragmatic architect. *IEEE Software*, pages 10–11, September 2009. (Cited on page 210.)

[Bus10] Frank Buschmann. Learning from failure, part iii: On hammers and nails, and falling in love with technology and design. *IEEE Software*, pages 49–51, March 2010. (Cited on page 212.)

[Bus11a] Frank Buschmann. Gardening your architecture, part 1: Refactoring. *IEEE Software*, pages 92–94, July 2011. (Cited on page 212.)

[Bus11b] Frank Buschmann. Tests: The architect's best friend. *IEEE Software*, pages 7–9, May 2011. (Cited on page 213.)

[Bus11c] Frank Buschmann. Unusable software is useless, part 2. *IEEE Software*, pages 100–102, March 2011. (Cited on pages 212 and 213.)

[Bus12] Frank Buschmann. To boldly go where no one has gone before. *IEEE Software*, pages 23–25, Jan 2012. (Cited on pages 212 and 213.)

[CB11] James O. Coplien and Gertrud Bjørnvig. *Lean architecture: for agile software development*. John Wiley & Sons, 2011. (Cited on page 64.)

[CFJ$^+$16] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering modeling languages: Turning domain knowledge into tools*. Chapman and Hall/CRC, 2016. (Cited on page 21.)

[CGB$^+$10] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting software architectures: views and beyond*. Addison-Wesley, Boston, 2nd edition, 2010. (Cited on pages 22, 34, 35, and 161.)

[CH10] Henrik B. Christensen and Klaus M. Hansen. An empirical investigation of architectural prototyping. *Journal of Systems and Software*, pages 133 – 142, 2010. SI: Top Scholars. (Cited on page 213.)

[Cha14] Kathy Charmaz. *Constructing grounded theory*. Sage, London, 2nd edition, 2014. (Cited on pages 59, 61, and 152.)

[Che] Checkstyle. Checkstyle. `https://checkstyle.sourceforge.io/`. Last access 31.10.2019. (Cited on page 71.)

[Cho03] Gobinda G. Chowdhury. Natural language processing. *Annual Review of Information Science and Technology*, pages 51–89, 2003. (Cited on page 191.)

[CHS09] Henrik B. Christensen, Klaus M. Hansen, and Kari R. Schougaard. An empirical study of software architects' concerns. In *Proceedings of the 16th Asia-Pacific Software Engineering Conference*, pages 111–118, Washington, DC, USA, Dec 2009. IEEE. (Cited on pages 212 and 213.)

[CKK$^+$03] Paul Clements, Rick Kazman, Mark Klein, et al. *Evaluating software architectures: Methods and Case studies*. Tsinghua University Press Beijing, 2003. (Cited on page 29.)

[CKS05] Andreas Christl, Rainer Koschke, and Margaret-Anne Storey. Equipping the reflexion method with automated clustering. In *12th Working Conference on Reverse Engineering (WCRE'05)*, page 10 pp., Nov 2005. (Cited on page 79.)

[Cle95] Paul C. Clements. Formal methods in describing architectures. Technical report, Carnegie-Mellon University Pittsburgh PA Software Engineering Institute, 1995. (Cited on page 31.)

[Cle96] Paul C. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, IWSSD '96, pages 16–25, Washington, DC, USA, 1996. IEEE Computer Society. (Cited on page 37.)

[CLN14] Andrea Caracciolo, Mircea Filip Lungu, and Oscar Nierstrasz. How do software architects specify and validate quality requirements? In *Proceedings of the 8th European Conference of Software Architecture*, pages 374–389. Springer International Publishing, Cham, 2014. (Cited on pages 104, 212, and 213.)

[CLN15] Andrea Caracciolo, Mircea F. Lungu, and Oscar Nierstrasz. A unified approach to architecture conformance checking. In *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture*, pages 41–50, Washington, DC, USA, May 2015. IEEE. (Cited on pages 23, 40, 82, 153, and 158.)

[CLvV07] Viktor Clerc, Patricia Lago, and Hans van Vliet. *The Architect's Mindset*, pages 231–249. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. (Cited on page 212.)

[cql] Cqlinq syntax. `https://www.ndepend.com/docs/cqlinq-syntax`. Last access 1.11.2019. (Cited on page 85.)

[CSW18] Nacha Chondamrongkul, Jing Sun, and Ian Warren. Ontology-based software architectural pattern recognition and reasoning. In *The 30th International Conference on Software Engineering and Knowledge Engineering*, pages 23–28, 2018. (Cited on page 100.)

[cuc] Cucumber. `https://cucumber.io/`. Last access 31.10.2019. (Cited on page 56.)

[DAB$^+$11] Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. Mse and famix 3.0: an interexchange format and source code model family. Technical report, 2011. (Cited on pages 124, 147, and 233.)

[DBFL$^+$07] Remco C. De Boer, Rik Farenhorst, Patricia Lago, Hans Van Vliet, Viktor Clerc, and Anton Jansen. Architectural knowledge: Getting to the core. In *International Conference on the Quality of Software Architectures*, pages 197–214. Springer, 2007. (Cited on page 99.)

[DCFKK09] Juri Luca De Coi, Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. *Controlled English for Reasoning on the Semantic Web*, pages 276–308. Springer Berlin Heidelberg, 2009. (Cited on pages 98 and 104.)

[DDC$^+$10] Ronald Denaux, Vania Dimitrova, Anthony G. Cohn, Catherine Dolbear, and Glen Hart. Rabbit to owl: Ontology authoring with a CNL-based tool. In Norbert E. Fuchs, editor, *Controlled Natural Language*, pages 246–264. Springer Berlin Heidelberg, 2010. (Cited on page 56.)

[dGTLK17] Klaas Andries de Graaf, Antony Tang, Peng Liang, and Ali Khalili. Querying software architecture knowledge as linked open data. In *IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017, Gothenburg, Sweden*, pages 272–277, 2017. (Cited on page 100.)

[dGTLvV12] Klaas Andries de Graaf, Antony Tang, Peng Liang, and Hans van Vliet. Ontology-based software architecture documentation. In *Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, Helsinki, Finland*, pages 121–130, 2012. (Cited on page 99.)

[DHHJ10] Florian Deissenboeck, Lars Heinemann, Benjamin Hummel, and Elmar Juergens. Flexible architecture conformance assessment with ConQAT. In *ACM/IEEE 32nd International Conference on Software Engineering*, pages 247–250, May 2010. (Cited on page 79.)

[DHK$^+$07] Cathy Dolbear, Glen Hart, Katalin Kovacs, John Goodwin, and Sheng Zhou. The rabbit language: description, syntax and conversion to OWL. *Ordnance Survey Research Labs Technical Report*, 2007. (Cited on page 107.)

[DHT01] Eric M. Dashofy, André Van der Hoek, and Richard N. Taylor. A highly-extensible, XML-based architecture description language. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, pages 103–112, Washington, DC, USA, 2001. IEEE Computer Society. (Cited on page 88.)

[DHT05] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Trans. Softw. Eng. Methodol.*, pages 199–245, 2005. (Cited on page 37.)

[DKL09] S. Duszynski, J. Knodel, and M. Lindvall. SAVE: Software architecture visualization and evaluation. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 323–324, March 2009. (Cited on pages 41 and 79.)

[DL13] Tom DeMarco and Tim Lister. *Peopleware: productive projects and teams.* Addison-Wesley, 2013. (Cited on page 21.)

[DLB14] Ana Dragomir, Horst Lichter, and Tiberiu Budau. Systematic architectural decision management, a process-based approach. In *IEEE/IFIP Conference on Software Architecture*, pages 255–258, April 2014. (Cited on page 35.)

[dMSV$^+$08] Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. *.QL: Object-Oriented Queries Made Easy*, pages 78–133. Springer Berlin Heidelberg, 2008. (Cited on page 85.)

[DP09] Stéphane Ducasse and Damian Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, pages 573–591, July 2009. (Cited on page 33.)

[DR99] Maarten De Rijke. *Handbook of Tableau Methods.* Number 4. Springer Netherlands, 1999. (Cited on page 48.)

[DSB12] Lakshitha De Silva and Dharini Balasubramaniam. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, pages 132–151, 2012. (Cited on pages 22 and 40.)

[EJ85] Agneta Eriksson and Anna-Lena Johansson. Neat explanation of proof trees. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1*, page 379–381. Morgan Kaufmann Publishers Inc., 1985. (Cited on page 141.)

[EP16] Murat Erder and Pierre Pureur. What's the architect's role in an agile, cloud-centric world? *IEEE Software*, pages 30–33, September 2016. (Cited on page 213.)

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005. (Cited on pages 99 and 131.)

[Eva04] Eric Evans. *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional, Boston, 2004. (Cited on pages 25, 185, and 225.)

[Fab10] Roland Faber. Architects as service providers. *IEEE Software*, pages 33–40, 2010. (Cited on page 213.)

[FdB06] Rik Farenhorst and Remco C. de Boer. Core concepts of an ontology of architectural design decisions. *Technical Report IR-IMSE-002, Dept. Computer Science*, 2006. (Cited on page 99.)

[FG10] Georg Fairbanks and David Garlan. *Just Enough Software Architecture: A Risk-driven Approach.* Marshall & Brainerd, Boulder, 2010. (Cited on pages 22, 39, and 139.)

[FGH06] Peter Feiler, David Gluch, and John Hudak. The architecture analysis & design language (AADL): An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2006. (Cited on page 88.)

[Fow99] *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. (Cited on page 153.)

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, Boston, 2002. (Cited on pages 63 and 139.)

[Fow03] Martin Fowler. Who needs an architect? *IEEE Software*, pages 11–13, September 2003. (Cited on page 21.)

[FTB$^+$07] Adam Funk, Valentin Tablan, Kalina Bontcheva, Hamish Cunningham, Brian Davis, and Siegfried Handschuh. CLOnE: Controlled language for ontology editing. In *Proceedings of the Sixth International and Second Asian Semantic Web Conference*, pages 142–155, 2007. (Cited on page 56.)

[Gó01] Asunción Gómez-Pérez. Evaluation of ontologies. *International Journal of Intelligent Systems*, pages 391–409, March 2001. (Cited on page 146.)

[GAMD16] Matthias Galster, Samuil Angelov, Marcel Meesters, and Philipp Diebold. A multiple case study on the architect's role in scrum. In *Product-Focused Software Process Improvement*, pages 432–447, Cham, 2016. Springer International Publishing. (Cited on page 213.)

[Gar00] David Garlan. Software architecture: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 91–101, 2000. (Cited on page 31.)

[Gar03] David Garlan. Formal modeling and analysis of software architecture: Components, connectors, and events. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 1–24. Springer, 2003. (Cited on pages 29 and 37.)

[Gas04] Susan Gasson. Rigor in grounded theory research: An interpretive perspective on generating theory from qualitative field studies. *The handbook of information systems research*, pages 79–102, 2004. (Cited on page 74.)

[GCSY08] Gokhan Gokyer, Semih Cetin, Cevat Sener, and Meltem T. Yondem. Non-functional requirements to architectural concerns: Ml and nlp at crossroads. In *The Third International Conference on Software Engineering Advances*, pages 400–406, Oct 2008. (Cited on page 212.)

[Ger] Gerrit. Gerrit code review. `https://www.gerritcodereview.com`. Last access 31.10.2019. (Cited on page 70.)

[GF95] Michael Grüninger and Mark S. Fox. Methodology for the Design and Evaluation of Ontologies. In *Workshop on Basic Ontological Issues in Knowledge Sharing, April 13, 1995*, 1995. (Cited on page 147.)

[git] Git. `https://git-scm.com/`. Last access 1.11.2019. (Cited on pages 96 and 134.)

[git19] CNL toolchain. `https://github.com/sandrellaella/cnl-main-repository`, 2019. (Cited on page 145.)

[GKB10] Bernhard Groene, Wolfram Kleis, and Jochen Boeder. Educating architects in industry - the sap architecture curriculum. In *17th IEEE International Conference and Workshops on Engineering of Computer Based Systems*, pages 201–205, March 2010. (Cited on page 212.)

[GKR$^+$14] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased modeling. *Proceedings of the 4th International Workshop on Software Language Engineering*, 2014. (Cited on page 35.)

[Gla78] Barney G. Glaser. *Theoretical sensitivity: Advances in the methodology of grounded theory.* Sociology Pr, 1978. (Cited on pages 24, 61, 151, 152, and 164.)

[GMW97] David Garlan, Robert Monroe, and David Wile. Acme: An architecture description interchange language. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '97, pages 7–. IBM Press, 1997. (Cited on page 88.)

[Gru95] Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing? *International Journal of Human-Computer Studies*, pages 907 – 928, 1995. (Cited on pages 22 and 24.)

[HA05] Siw E. Hove and Bente Anda. Experiences from conducting semi-structured interviews in empirical software engineering research. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 10–23, Washington, DC, Sept 2005. IEEE. (Cited on page 59.)

[hel] hellow2morrow. hello2morrow - sotograph. `https://www.hello2morrow.com/products/sotograph`. Last access 31.10.2019. (Cited on page 73.)

[Her11] Sebastian Herold. *Architectural Compliance in Component-Based Systems*. PhD thesis, Clausthal University of Technology, 2011. (Cited on pages 40, 84, and 226.)

[HH06] Daqing Hou and H. James Hoover. Using SCL to specify and check design intent in source code. *IEEE Transactions On Software Engineering*, pages 404–423, June 2006. (Cited on page 83.)

[HKN$^+$07] Christine Hofmeister, Philippe Kruchten, Robert L. Nord, Henk Obbink, Alexander Ran, and Pierre America. A general model of software architecture design derived from five industrial approaches. *Journal of Systems and Software*, pages 106–126, 2007. (Cited on page 32.)

[HKR09] Pascal Hitzler, Markus Krötzsch, and Sebastian Rudolph. *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC, 1st edition, 2009. (Cited on pages 49 and 54.)

[HKS06] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible sroiq. In *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning*, KR'06, pages 57–67. AAAI Press, 2006. (Cited on page 43.)

[HM14] Sebastian Herold and Matthias Mair. Recommending refactorings to re-establish architectural consistency. In *Software Architecture - 8th European Conference, Vienna, Austria*, pages 390–397, 2014. (Cited on page 192.)

[HM16] Rashina Hoda and Latha K. Murugesan. Multi-level agile project management challenges: A self-organizing team perspective. *Journal of Systems and Software*, pages 245 – 257, 2016. (Cited on page 61.)

[HMRS13] Sebastian Herold, Matthias Mair, Andreas Rausch, and Ingrid Schindler. Checking conformance with reference architectures: A case study. In *17th IEEE International Enterprise Distributed Object Computing Conference*, pages 71–80, September 2013. (Cited on page 57.)

[HNS00] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied software architecture*. Addison-Wesley Professional, Boston, 2000. (Cited on pages 29 and 35.)

[Hor51] Alfred Horn. On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, page 14–21, 1951. (Cited on page 53.)

[HPS10] Matthew Horridge, Bijan Parsia, and Ulrike Sattler. Justification oriented proofs in OWL. In *The Semantic Web – ISWC 2010*, pages 354–369, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. (Cited on pages 117 and 141.)

[HS] Peter Hruschka and Gernot Starke. arc42 - ressourcen für architekten. `https://arc42.de/`. Last access 31.10.2019. (Cited on page 36.)

[HS06] Hans-Jörg Happel and Stefan Seedorf. Applications of ontologies in software engineering. In *Proceedings of Workshop on Sematic Web Enabled Software Engineering" on the ISWC*, pages 5–9. Citeseer, 2006. (Cited on page 99.)

[HSBA10] Maryam Hasan, Eleni Stroulia, Denilson Barbosa, and Manar Alalfi. Analyzing natural-language artifacts of the software process. In *IEEE International Conference on Software Maintenance*, pages 1–5, September 2010. (Cited on page 21.)

[HST99] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical reasoning for expressive description logics. In *Logic for Programming and Automated Reasoning*, pages 161–180. Springer Berlin Heidelberg, 1999. (Cited on page 47.)

[IEE00] IEEE recommended practice for architectural description for software-intensive systems. *IEEE Std 1471-2000*, pages 1–30, Oct 2000. (Cited on pages 30 and 35.)

[IEE11] ISO/IEC/IEEE systems and software engineering – architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pages 1–46, Dec 2011. (Cited on page 35.)

[ISO11] ISO. Road vehicles – Functional safety, 2011. (Cited on page 65.)

[isya] Isyfact jQassistant Plugin - Batch. `https://github.com/IsyFact/isyfact-jqassistant-plugin/blob/master/src/main/resources/META-INF/jqassistant-rules/batch.xml`. Last access 31.08.2019. (Cited on page 223.)

[isyb] Isyfact jQassistant Plugin - Fehlerbehandlung. `https://github.com/IsyFact/isyfact-jqassistant-plugin/blob/master/src/main/resources/META-INF/jqassistant-rules/fehlerbehandlung.xml`. Last access 31.08.2019. (Cited on pages 224 and 226.)

[isyc] Isyfact jQassistant Plugin - Logging. `https://github.com/IsyFact/isyfact-jqassistant-plugin/blob/master/src/main/resources/META-INF/jqassistant-rules/logging.xml`. Last access 31.08.2019. (Cited on page 225.)

[isyd] Isyfact jQassistant Plugin - Persistence. `https://github.com/IsyFact/isyfact-jqassistant-plugin/blob/master/src/main/resources/META-INF/jqassistant-rules/persistence.xml`. Last access 31.08.2019. (Cited on pages 224 and 225.)

[isye] Isyfact jQassistant Plugin - Service. `https://github.com/IsyFact/isyfact-jqassistant-plugin/blob/master/src/main/resources/META-INF/jqassistant-rules/service.xml`. Last access 31.08.2019. (Cited on pages 226 and 227.)

[isyf] Isyfact jQassistant Plugin - Überwachung. `https://github.com/IsyFact/isyfact-jqassistant-plugin/blob/master/src/main/resources/META-INF/jqassistant-rules/ueberwachung.xml`. Last access 31.08.2019. (Cited on page 224.)

[jab] Jabref. `http://www.jabref.org/`. Last access 1.11.2019. (Cited on pages 150 and 168.)

[Jac12] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2012. (Cited on page 22.)

[jar] JDK 6 Java Archive (JAR) - related apis and developer guides. `https://docs.oracle.com/javase/6/docs/technotes/guides/jar/jar.html/#JARIndex`. Last access 1.11.2019. (Cited on page 131.)

[Jas20] Stefanie Jasser. Enforcing architectural security decisions. In *2020 IEEE International Conference on Software Architecture (ICSA), 2020, Salvador, Brasil, March 16-20*, pages 35–45. IEEE, 2020. (Cited on page 155.)

[jav] Javadoc tool home page. `https://www.oracle.com/technetwork/java/javase/documentation/javadoc-137458.html`. Last access 1.11.2019. (Cited on page 133.)

[JAvdV09] Anton Jansen, Paris Avgeriou, and Jan Salvador van der Ven. Enriching software architecture documentation. *Journal of Systems and Software*, pages 1232–1248, 2009. (Cited on page 34.)

[JB05] Anton Jansen and Jan Bosch. Software architecture as a set of architectural design decisions. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pages 109–120, Washington, DC, 2005. IEEE. (Cited on page 31.)

[jee] Java Platform, Enterprise Edition (Java EE) - Oracle Technology Network. `https://www.oracle.com/java/technologies/java-ee-glance.html`. Last access 1.11.2019. (Cited on page 139.)

[jen] Apache jena. `https://jena.apache.org/`. Last access 1.11.2019. (Cited on page 146.)

[JM00] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000. (Cited on page 23.)

[jme] JMeter. `https://jmeter.apache.org/`. Last access 1.11.2019. (Cited on page 83.)

[jqa] jQassistant - Your Software. `https://jqassistant.org/`. Last access 1.11.2019. (Cited on page 85.)

[KAZ18] Oliver Kopp, Anita Armbruster, and Olaf Zimmermann. Markdown architectural decision records: Format and tool support. In *Proceedings of the 10th Central European Workshop on Services and their Composition, Dresden, Germany, February 8-9*, pages 55–62, 2018. (Cited on page 25.)

[KB14] Tobias Kuhn and Alexandre Bergel. Verifiable source code documentation in controlled natural language. *Science of Computer Programming*, pages 121 – 140, 2014. Special issue on Advances in Smalltalk based Systems. (Cited on page 150.)

[KC04] Graham Klyne and Jeremy J. Carroll. Resource description framework (RDF): Concepts and abstract syntax. W3C Recommendation, 2004. (Cited on page 49.)

[KC07] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. In *EBSE Technical Report*. January 2007. (Cited on pages 58 and 209.)

[KDV07] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on page 21.)

[Kem93] Chris F. Kemerer. Reliability of function points measurement: A field experiment. *Commun. ACM*, pages 85–97, February 1993. (Cited on page 169.)

[KGT05] Andreas Knöpfel, Bernhard Gröne, and Peter Tabeling. *Fundamental modeling concepts: Effective Communication of IT Systems*. Wiley, England, 2005. (Cited on page 67.)

[Kle05] John Klein. How does the architect's role change as the software ages? In *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 141–141, November 2005. (Cited on page 213.)

[Kle08] Anneke Kleppe. *Software language engineering: creating domain-specific languages using metamodels*. Pearson Education, 2008. (Cited on page 107.)

[Kle16] John Klein. What makes an architect successful? *IEEE Software*, pages 20–22, 2016. (Cited on page 213.)

[KLvV06] Philippe Kruchten, Patricia Lago, and Hans van Vliet. Building up and reasoning about architectural knowledge. In *Quality of Software Architectures*. Springer Berlin Heidelberg, 2006. (Cited on page 31.)

[KN16] Jens Knodel and Matthias Naab. *Pragmatic Evaluation of Software Architectures*. Springer Publishing Company, Incorporated, 1st edition, 2016. (Cited on pages 22, 29, 40, 41, and 145.)

[Krö10a] Markus Krötzsch. *Description logic rules*, volume 8. IOS Press, 2010. (Cited on page 53.)

[Krö10b] Markus Krötzsch. Efficient inferencing for OWL EL. In *Logics in Artificial Intelligence*, pages 234–246. Springer Berlin Heidelberg, 2010. (Cited on page 141.)

[Kru99] Philippe Kruchten. *The Software Architect*, pages 565–583. Springer US, Boston, MA, 1999. (Cited on pages 212 and 213.)

[Kru00] Philippe Kruchten. *The Rational Unified Process: An Introduction, Second Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000. (Cited on pages 21 and 38.)

[Kru04a] Philippe Kruchten. An ontology of architectural design decisions in software intensive systems. In *2nd Groningen workshop on software variability*, pages 54–61, 2004. (Cited on page 99.)

[Kru04b] Philippe Kruchten. *The rational unified process: an introduction*. Addison-Wesley Professional, 2004. (Cited on pages 29, 31, and 35.)

[Kru08] Philippe Kruchten. What do software architects really do? *Journal of Systems and Software*, 81(12):2413 – 2416, 2008. (Cited on pages 21 and 213.)

[Kru09] Philippe Kruchten. Documentation of software architecture from a knowledge management perspective–design representation. In *Software Architecture Knowledge Management*, pages 39–57. Springer, 2009. (Cited on pages 34 and 35.)

[KSH12] Markus Krötzsch, Frantisek Simancik, and Ian Horrocks. A description logic primer. *CoRR*, 2012. (Cited on pages 42, 43, 44, 45, 46, 48, and 51.)

[Kuh10] Tobias Kuhn. *Controlled English for Knowledge Representation*. PhD thesis, Faculty of Economics, Business Administration and Information Technology of the University of Zurich, 2010. (Cited on pages 23, 56, and 107.)

[Kuh13] Tobias Kuhn. The understandability of OWL statements in controlled english. *Semantic Web*, pages 101–115, January 2013. (Cited on pages 56, 98, and 150.)

[Kuh14] Tobias Kuhn. A survey and classification of controlled natural languages. *Computational Linguistics*, pages 121–170, 2014. (Cited on pages 25, 55, and 150.)

[KWWB03] Anneke G Kleppe, Jos Warmer, Jos B Warmer, and Wim Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003. (Cited on pages 34 and 107.)

[KZ10] Patrick Könemann and Olaf Zimmermann. Linking design decisions to design models in model-based software development. In Muhammad Ali Babar and Ian Gorton, editors, *Software Architecture*, pages 246–262, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. (Cited on page 23.)

[Lag14] Robert Lagerstedt. Using automated tests for communicating and verifying non-functional requirements. In *IEEE 1st International Workshop on Requirements Engineering and Testing (RET)*, pages 26–28, August 2014. (Cited on pages 212 and 213.)

[lat] Lattix architect. `https://www.lattix.com/products-architecture-issues/#architect`. Last access 1.11.2019. (Cited on page 82.)

[LMK15] Angela Lozano, Kim Mens, and Andy Kellens. Usage contracts: Offering immediate feedback on violations of structural source-code regularities. *Science of Computer Programming*, pages 73 – 91, 2015. (Cited on page 85.)

[LMM$^+$15] Patricia Lago, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Antony Tang. The road ahead for architectural languages. *IEEE Software*, pages 98–105, Jan 2015. (Cited on pages 37, 88, and 89.)

[LN13] François Lévy and Adeline Nazarenko. Formalization of natural language regulations through sbvr structured english. In *Theory, Practice, and Applications of Rules on the Web*, pages 19–33. Springer Berlin Heidelberg, 2013. (Cited on page 56.)

[LSF03] Timothy C Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: the state of the practice. *IEEE Software*, pages 35–39, Nov 2003. (Cited on page 79.)

[LV95] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, pages 717–734, Sep. 1995. (Cited on page 41.)

[LVM95] David C. Luckham, James Vera, and Sigurd Meldal. Three concepts of system architecture. Technical report, Stanford, CA, USA, 1995. (Cited on page 87.)

[mac] Macker. `https://innig.net/macker/`. Last access 1.11.2019. (Cited on pages 82 and 170.)

[man12] OWL 2 Web Ontology Language Manchester Syntax (Second Edition). `https://www.w3.org/TR/owl2-manchester-syntax/`, December 2012. (Cited on pages 51 and 98.)

[mav] Maven – Welcome to Apache Maven. `https://maven.apache.org/`. Last access 1.11.2019. (Cited on page 131.)

[MB09] Philipp Mayring and Eva Brunner. *Qualitative Inhaltsanalyse*, pages 669–680. Gabler, Wiesbaden, 2009. (Cited on page 24.)

[McB04] Matthew R. McBride. The software architect: Essence, intuition, and guiding principles. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '04, pages 230–235, New York, USA, 2004. ACM. (Cited on pages 212 and 213.)

[McB07] Matthew R. McBride. The software architect. *Commun. ACM*, 50(5):75–81, May 2007. (Cited on page 21.)

[MCH16] Mehdi Mirakhorli and J. Cleland-Huang. Detecting, tracing, and monitoring architectural tactics in code. *IEEE Transactions on Software Engineering*, pages 205–220, March 2016. (Cited on page 23.)

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag. (Cited on page 88.)

[MEM$^+$13] Ralf Mitschke, Michael Eichberg, Mira Mezini, Alessandro Garcia, and Isela Macia. Modular specification and checking of structural dependencies. In *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development*, pages 85–96, New York, USA, 2013. ACM. (Cited on page 83.)

[Men00] Kim Mens. *Automating architectural conformance checking by means of logic meta programming.* PhD thesis, Vrije Universiteit Brussel, 2000. (Cited on page 83.)

[Mer09] Bernhard Merkle. Stopping (and reversing) the architectural erosion of software systems. an industrial case study. *Software Engineering*, 2009. (Cited on page 40.)

[MFC$^+$18] Yuzhan Ma, Sarah Fakhoury, Michael Christensen, Venera Arnaoudova, Waleed Zogaan, and Mehdi Mirakhorli. Automatic classification of software artifacts in open-source applications. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 414–425, New York, USA, 2018. ACM. (Cited on page 120.)

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, pages 316–344, December 2005. (Cited on page 38.)

[MIS] MISRA. Misra - the motor industry software reliability association. `https://www.autosar.org/`. Last access 31.10.2019. (Cited on page 65.)

[MLM$^+$13] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering*, pages 869–891, June 2013. (Cited on pages 23, 37, 78, and 88.)

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '95, pages 18–28, New York, USA, 1995. ACM. (Cited on page 79.)

[moo] Moose. `http://moosetechnology.org/`. Last access 1.11.2019. (Cited on pages 83 and 158.)

[MPB14] Antonio Martini, Lars Pareto, and Jan Bosch. *Role of Architects in Agile Organizations*, pages 39–50. Springer International Publishing, Cham, 2014. (Cited on pages 212 and 213.)

[MR97] Nenad Medvidovic and David S. Rosenblum. Domains of concern in software architectures and architecture description languages. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL)*, page 199–212, Berkeley, CA, USA, 1997. USENIX Association. (Cited on page 37.)

[MSB$^+$14] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014. (Cited on pages 106 and 107.)

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, pages 70–93, Jan 2000. (Cited on pages 22, 37, and 88.)

[Mus15] Mark A Musen. The protégé project: A look back and a look forward. *AI matters*, pages 4–12, June 2015. (Cited on page 98.)

[NCGA14] Paul Brillant Feuto Njonko, Sylviane Cardey, Peter Greenfield, and Walid El Abed. Rulecnl: A controlled natural language for business rule specifications. *CoRR*, 2014. (Cited on page 56.)

[neo] Neo4j graph platform – the leader in graph databases. `https://neo4j.com/`. Last access 1.11.2019. (Cited on page 85.)

[Nic18] Ana Nicolaescu. *Behavior-Based Architecture Conformance Checking.* PhD thesis, RWTH Aachen University, Germany, 2018. (Cited on pages 23 and 41.)

[NM$^+$01] Natalya F. Noy, Deborah L McGuinness, et al. Ontology development 101: A guide to creating your first ontology, 2001. (Cited on page 122.)

[NS07] David Nadeau and Satoshi Sekine. A survey of named entity recognition and classification. *Lingvisticae Investigationes*, pages 3–26, 2007. (Cited on page 191.)

[Obj06] Object Management Group (OMG). Meta-Object Facility (MOF) Specification, Version 2.0. OMG Document Number formal/2006-01-01 (`http://www.omg.org/spec/MOF/2.0`), 2006. (Cited on page 36.)

[OC08] Mark O'Keeffe and Mel Ó Cinnéide. Search-based refactoring for software maintenance. *Journal of Systems and Software*, pages 502 – 516, 2008. Selected papers from the 10th Conference on Software Maintenance and Reengineering (CSMR 2006). (Cited on page 192.)

[ocl14] About the Object Constraint Language specification version 2.4, 2014. (Cited on page 89.)

[OEW18] Tobias Olsson, Morgan Ericsson, and Anna Wingkvist. Towards improved initial mapping in semi automatic clustering. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings, Madrid, Spain, September 24-28, 2018*, pages 51:1–51:7, 2018. (Cited on page 79.)

[OK13] Mert Ozkaya and Christos Kloukinas. Are we there yet? analyzing architecture description languages for formal analysis, usability, and realizability. In *39th Euromicro Conference on Software Engineering and Advanced Applications*, pages 177–184, Sep. 2013. (Cited on page 78.)

[owl03] OWL web ontology language xml presentation syntax. `https://www.w3.org/TR/owl-xmlsyntax/`, June 2003. Last access: 31-10-2019. (Cited on page 51.)

[owl12a] OWL 2 web ontology language document overview (second edition). `https://www.w3.org/TR/owl2-overview/`, December 2012. (Cited on page 49.)

[owl12b] OWL 2 web ontology language structural specification and functional-style syntax (second edition). `https://www.w3.org/TR/owl2-syntax/`, December 2012. Last access: 31-10-2019. (Cited on page 51.)

[Ozk18a] Mert Ozkaya. The analysis of architectural languages for the needs of practitioners. *Software: Practice and Experience*, pages 985–1018, 2018. (Cited on page 37.)

[Ozk18b] Mert Ozkaya. Do the informal & formal software modeling notations satisfy practitioners for software architecture modeling? *Information and Software Technology*, pages 15 – 33, 2018. (Cited on pages 37, 78, and 88.)

[Par11] Fernando Silva Parreiras. *Marrying model-driven engineering and ontology technologies: The TwoUse approach.* PhD thesis, University of Koblenz-Landau, 2011. (Cited on page 100.)

[PB01] Daniel J. Paulish and Len Bass. *Architecture-centric software project management: A practical guide.* Addison-Wesley Longman Publishing Co., Inc., 2001. (Cited on page 31.)

[PEE12] Lars Pareto, Peter Eriksson, and Staffan Ehnebom. Concern coverage in base station development: an empirical investigation. *Software & Systems Modeling*, pages 409–429, July 2012. (Cited on page 212.)

[Pen03] Tom Pender. *UML bible.* John Wiley & Sons, Inc., 2003. (Cited on page 126.)

[PGH09] Claus Pahl, Simon Giesecke, and Wilhelm Hasselbring. Ontology-based modelling of architectural styles. *Information & Software Technology*, pages 1739–1749, 2009. (Cited on page 100.)

[PKB13] Leo Pruijt, Christian Koppe, and Sjaak Brinkkemper. Architecture compliance checking of semantically rich modular architectures: A comparative study of tool support. In *Proceedings of the International Conference on Software Maintenance*, pages 220–229. IEEE, 2013. (Cited on pages 82 and 83.)

[PKvdWB14] Leo J. Pruijt, Christian Köppe, Jan Martijn van der Werf, and Sjaak Brinkkemper. Husacct: Architecture compliance checking with rich sets of module and rule types. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 851–854, New York, NY, USA, 2014. ACM. (Cited on pages 57, 82, 153, 158, and 172.)

[PKvdWB17] Leo Pruijt, Christian Köppe, Jan Martijn van der Werf, and Sjaak Brinkkemper. The accuracy of dependency analysis in static architecture compliance checking. *Software: Practice and Experience*, pages 273–309, 2017. (Cited on page 176.)

[pmd] PMD. `https://pmd.github.io/`. Last access 1.11.2019. (Cited on page 83.)

[pro] Protégé. `https://protege.stanford.edu/`. Last access 1.11.2019. (Cited on page 98.)

[PSV94] Dewayne E. Perry, Nancy A. Staudenmayer, and Lawrence G. Votta. People, organizations, and process improvement. *IEEE Software*, pages 36–45, July 1994. (Cited on page 21.)

[PTV$^+$10] Leonardo Passos, Ricardo Terra, Marco T. Valente, Renato Diniz, and Nabor das Chagas Mendonca. Static architecture-conformance checking: An illustrative overview. *IEEE Software*, pages 82–89, Sept 2010. (Cited on pages 79 and 88.)

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, pages 40–52, October 1992. (Cited on page 35.)

[PZ10] Frances Paulisch and Peter Zimmerer. A role-based qualification and certification program for software architects: an experience report from siemens. In *ACM/IEEE 32nd International Conference on Software Engineering*, pages 21–27, May 2010. (Cited on page 213.)

[Rat09] Daniel Petrica Ratiu. *Intentional meaning of programs.* PhD thesis, Technische Universität München, 2009. (Cited on page 123.)

[rdf14] RDF 1.1 XML syntax. `https://www.w3.org/TR/rdf-syntax-grammar/`, February 2014. Last access: 31-10-2019. (Cited on page 51.)

[Req] Reqtify. Reqtify - claytex. `https://www.claytex.com/products/reqtify/`. Last access 31.10.2019. (Cited on page 71.)

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2Nd Edition).* Pearson Higher Education, 2004. (Cited on pages 21, 67, and 89.)

[RLGBAB08] Jacek Rosik, Andrew Le Gear, Jim Buckley, and Muhammad Ali Babar. An industrial case study of architecture conformance. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 80–89, New York, USA, 2008. ACM. (Cited on page 41.)

[RR00] Pierre N. Robillard and Martin P. Robillard. Types of collaborative work in software engineering. *Journal of Systems and Software*, pages 219 – 224, 2000. (Cited on page 21.)

[RW11] Nick Rozanski and Eóin Woods. *Software systems architecture: working with stakeholders using viewpoints and perspectives.* Addison-Wesley, 2011. (Cited on page 30.)

[Sau10] Joachim Sauer. *Architecture-Centric Development in Globally Distributed Projects*, pages 321–329. Springer Berlin Heidelberg, 2010. (Cited on pages 212 and 213.)

[SB01] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001. (Cited on pages 21 and 38.)

[SB05] Jocelyn Simmonds and M. Cecilia Bastarrica. Description logics for consistency checking of architectural features in UML 2.0 models. *Departamento de Ciencias de la Computacion, Facultad de Ciencias Fısicas y Matematicas, Universidad de Chile*, 2005. (Cited on page 100.)

[SC$^+$90] Anselm Strauss, Juliet Corbin, et al. *Basics of qualitative research*, volume 15. Sage, Newbury Park, CA, 1990. (Cited on pages 24, 58, and 61.)

[Sch02] Rolf Schwitter. English as a formal specification language. In *Proceedings. 13th International Workshop on Database and Expert Systems Applications*, pages 228–232. IEEE, 2002. (Cited on pages 55 and 104.)

[Sch10] Rolf Schwitter. Controlled natural languages for knowledge representation. In *Proceedings of the 23rd International Conference on Computational Linguistics: Posters*, COLING '10, pages 1113–1121, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics. (Cited on page 25.)

[sek] International conference on software engineering and knowledge engineering. `http://ksiresearchorg.ipage.com/seke/seke19.html`. Last access 1.11.2019. (Cited on page 99.)

[sem] W3C Semantic Web Activity Homepage. `https://www.w3.org/2001/sw/`. Last access 1.11.2019. (Cited on page 98.)

[SH15] Sofia Sherman and Irit Hadar. Toward defining the role of the software architect: An examination of the soft aspects of this role. In *Proceedings of the Eighth International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '15, pages 71–76, Piscataway, NJ, USA, 2015. IEEE Press. (Cited on pages 212 and 213.)

[Sir10] Evren Sirin. Data validation with OWL integrity constraints. In *Web Reasoning and Rule Systems*, pages 18–22. Springer Berlin Heidelberg, 2010. (Cited on page 53.)

[SJSJ05] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 167–176, New York, 2005. ACM. (Cited on page 83.)

[SKC$^+$08] Rolf Schwitter, Kaarel Kaljur, Anne Cregan, Catherine Dolbear, and Glen Hart. A comparison of three controlled natural languages for OWL 1.1. In *In 4th OWL Experiences and Directions Workshop (OWLED 2008 DC*, 2008. (Cited on page 104.)

[SM17] Miroslaw Staron and Wilhelm Meding. A portfolio of internal quality metrics for software architects. In *Software Quality. Complexity and Challenges of Software Engineering in Emerging Technologies*, pages 57–69. Springer International Publishing, 2017. (Cited on page 212.)

[Sma14] John F. Smart. *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*. Manning Publications, 2014. (Cited on page 164.)

[Smo02] Kari Smolander. Four metaphors of architecture in software organizations: finding out the meaning of architecture in practice. In *Proceedings International Symposium on Empirical Software Engineering*, pages 211–221, Oct 2002. (Cited on pages 22, 31, 33, and 192.)

[Sol] Business Rule Solutions. Rulespeak. `http://www.rulespeak.com/en/`. Last access 31.10.2019. (Cited on page 56.)

[Son] Sonarqube. Code quality and security - sonarqube. `sonarqube.org`. Last access 31.10.2019. (Cited on page 71.)

[son17] Sonargraph-architect, 2017. (Cited on page 79.)

[Sow] John F. Sowa. Common logic controlled english. `http://www.jfsowa.com/clce/specs.htm`. Last access 31.10.2019. (Cited on page 56.)

[SPG$^+$07] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2):51 – 53, 2007. Software Engineering and the Semantic Web. (Cited on page 53.)

[spr] Spring. `https://spring.io/`. Last access 1.11.2019. (Cited on page 139.)

[sta] Stardog: The enterprise knowledge graph platform. `https://www.stardog.com/`. Last access 1.11.2019. (Cited on pages 145 and 146.)

[Stö17] Harald Störrle. How are conceptual models used in industrial software development?: A descriptive survey. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, EASE'17, pages 160–169, New York, NY, USA, 2017. ACM. (Cited on pages 21 and 36.)

[str18] Structure 101. `https://structure101.com/`, December 2018. (Cited on page 79.)

[SUS14] Sofia Sherman and Naomi Unkelos-Shpigel. What do software architects think they (should) do? In Lazaros Iliadis, Michael Papazoglou, and Klaus Pohl, editors, *Advanced Information Systems Engineering Workshops*, pages 219–225, Cham, 2014. Springer International Publishing. (Cited on pages 212 and 213.)

[SWGP10] Steffen Staab, Tobias Walter, Gerd Gröner, and Fernando Silva Parreiras. *Model Driven Engineering with Ontology Technologies*, pages 62–98. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. (Cited on page 100.)

[swr04] SWRL: A Semantic Web Rule Language Combining OWL and RuleML, May 2004. (Cited on pages 28, 53, and 121.)

[TAJ$^+$10] Antony Tang, Paris Avgeriou, Anton Jansen, Rafael Capilla, and Muhammad Ali Babar. A comparative study of architecture knowledge management tools. *Journal of Systems and Software*, pages 352–370, 2010. (Cited on page 32.)

[TBGH05] Antony Tang, Muhammad A. Babar, Ian Gorton, and Jun Han. A survey of the use and documentation of architecture design rationale. In *5th Working IEEE/IFIP Conference on Software Architecture*, pages 89–98, November 2005. (Cited on page 78.)

[tea] TEAMMATES - online peer feedback-evaluation system for student team projects. `https://teammatesv4.appspot.com/`. Last access 1.11.2019. (Cited on pages 150 and 168.)

[Tin10] Kai Ming Ting. *Precision and Recall*, pages 781–781. Springer US, Boston, MA, 2010. (Cited on page 150.)

[TKB10] Jonas Tappolet, Christoph Kiefer, and Abraham Bernstein. Semantic web enabled software analysis. *Web Semantics: Science, Services and Agents on the World Wide Web*, 8(2-3), 2010. (Cited on page 101.)

[TKF16] Damian A. Tamburri, Rick Kazman, and Hamed Fahimi. The architect's role in community shepherding. *IEEE Software*, pages 70–79, Nov 2016. (Cited on page 213.)

[TKLVV15] Damian A. Tamburri, Philippe Kruchten, Patricia Lago, and Hans Van Vliet. Social debt in software engineering: insights from industry. *Journal of Internet Services and Applications*, page 10, 2015. (Cited on page 21.)

[TMD09] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practice.* Wiley Publishing, 2009. (Cited on pages 35, 38, and 40.)

[tur08] Turtle - Terse RDF triple language, W3C team submission. `https://www.w3.org/TR/turtle/`, 2008. Last access: 31.10.2019. (Cited on pages 51 and 141.)

[TV09] Ricardo Terra and Marco Tulio Valente. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, pages 1073–1094, 2009. (Cited on pages 23, 57, 82, 153, and 158.)

[TVdH07] Richard N. Taylor and Andre Van der Hoek. Software design and architecture the once and future focus of software engineering. In *Future of Software Engineering (FOSE'07)*, pages 226–243. IEEE, 2007. (Cited on pages 30 and 31.)

[UD10] Hataichanok Unphon and Yvonne Dittrich. Software architecture awareness in long-term software product evolution. *Journal of Systems and Software*, pages 2211 – 2226, 2010. Interplay between Usability Evaluation and Software Development. (Cited on pages 212 and 213.)

[uml] Welcome to UML web site. `http://uml.org/`. Last access 1.11.2019. (Cited on page 89.)

[UNT10] Naoyasu Ubayashi, Jun Nomura, and Tetsuo Tamai. Archface: a contract place where architectural design and code meet together. In *ACM/IEEE 32nd International Conference on Software Engineering*, pages 75–84, May 2010. (Cited on page 87.)

[Vö10] Markus Völter. Architecture as language. *IEEE Software*, pages 56–64, 03 2010. (Cited on pages 22, 25, 31, 32, 33, 38, 89, 92, and 192.)

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, pages 26–36, 2000. (Cited on page 38.)

[VDSMSJ03] Ragnhild Van Der Straeten, Tom Mens, Jocelyn Simmonds, and Viviane Jonckers. Using description logic to maintain consistency between uml models. In *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*, pages 326–340. Springer Berlin Heidelberg, 2003. (Cited on page 100.)

[vHRH$^+$09] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey, and Dennis Kieselhorst. Continuous monitoring of software services: Design and application of the Kieker framework. 2009. (Cited on page 41.)

[Völ05] Markus Völter. A pattern language for building sustainable software architectures. *Computer methods and programs in biomedicine*, 2005. (Cited on pages 32 and 33.)

[W3Ca] W3C. RDF 1.1 concepts and abstract syntax. `https://www.w3.org/TR/rdf11-concepts/`. Last access 31.10.2019. (Cited on page 49.)

[W3Cb] W3C. RDF schema 1.1. `https://www.w3.org/TR/rdf-schema/`. Last access 31.10.2019. (Cited on page 49.)

[W3Cc] W3C. SPARQL query language for RDF. (Cited on page 54.)

[WB12] Rainer Weinreich and Georg Buchgeher. Towards supporting the software architecture life cycle. *Journal of Systems and Software*, pages 546–561, 2012. (Cited on page 32.)

[WB14] Rainer Weinreich and Georg Buchgeher. Automatic reference architecture conformance checking for soa-based software systems. In *IEEE/IFIP Conference on Software Architecture*, pages 95–104, April 2014. (Cited on pages 163, 223, 227, 228, and 229.)

[Wei71] Gerald M. Weinberg. *The psychology of computer programming.* Van Nostrand Reinhold New York, 1971. (Cited on page 21.)

[WGH$^+$12] Michael Würsch, Giacomo Ghezzi, Matthias Hert, Gerald Reif, and Harald C. Gall. SEON: a pyramid of ontologies for software evolution and its applications. *Computing*, pages 857–885, Nov 2012. (Cited on page 101.)

[WH05] Eoin Woods and Rich Hilliard. Architecture description languages in practice session report. In *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 243–246, November 2005. (Cited on pages 22, 37, 77, and 91.)

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003. (Cited on page 84.)

[Woh14] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, pages 38:1–38:10, New York, NY, USA, 2014. ACM. (Cited on page 211.)

[Woo15] Eoin Woods. Aligning architecture work with agile teams. *IEEE Software*, pages 24–26, Sept 2015. (Cited on pages 212 and 213.)

[WPT14] Sandra Williams, Richard Power, and Allan Third. How easy is it to learn a controlled natural language for building a knowledge base? In *Controlled Natural Language*, pages 20–32, Cham, 2014. Springer International Publishing. (Cited on page 150.)

[WRH$^+$12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering.* Springer Science & Business Media, 2012. (Cited on pages 159, 162, and 167.)

[WS09] Colin White and Rolf Schwitter. An update on PENG light. In *Proceedings of the Australasian Language Technology Association Workshop 2009*, pages 80–88, 2009. (Cited on page 56.)

[WSPS09] Tobias Walter, Fernando Silva Parreiras, and Steffen Staab. OntoDSL: An ontology-based framework for domain-specific languages. In *Model Driven Engineering Languages and Systems*, pages 408–422, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. (Cited on page 100.)

[xsd] OWL XML syntax - XML schemas. `https://www.w3.org/TR/owl-xmlsyntax/apd-schema.html`. Last access 1.11.2019. (Cited on page 129.)

[xtea] Xtend - modernized Java. `https://www.eclipse.org/xtend/`. Last access 1.11.2019. (Cited on page 146.)

[xteb] Xtext - language engineering made easy! `https://www.eclipse.org/Xtext/`. (Cited on pages 114 and 146.)

[Zim09] Olaf Zimmermann. *An architectural decision modeling framework for service oriented architecture design.* PhD thesis, 2009. (Cited on pages 21, 23, and 38.)

Eidesstattliche Versicherung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Dissertationsschrift selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ort, Datum	Sandra Schröder