Formalizing Architectural Rules with Ontologies - An Industrial Evaluation

1st Sandra Schröder
Software Engineering and Construction Methods
Universität Hamburg
Hamburg, Germany
schroeder@informatik.uni-hamburg.de

2nd Georg Buchgeher
Software Competence Center Hagenberg GmbH
Hagenberg im Mühlkreis, Austria
georg.buchgeher@scch.at

***Abstract*—Architecture conformance checking is an important means for quality control to assess that the system implementation adheres to its defined software architecture. Ideally, this process is automated to support continuous quality control. Many different approaches exist for automated conformance checking. However, these approaches are often limited in terms of supported concepts for describing and analyzing software architectures. We have developed an ontology-based approach that seeks to overcome the limited expressiveness of existing approaches. As a frontend of the formalism, we provide a Controlled Natural Language. In this paper, we present an industrial validation of the approach. For this, we collected architectural rules from three industrial projects. In total, we discovered 56 architectural rules in the projects. We successfully formalized 80% of those architectural rules. Additionally, we discussed the formalization with the corresponding software architect of each project. We found that the original intention of each architectural rule is properly reflected in the formalization. The results of the study show that projects could greatly benefit from applying an ontology-based approach, since it helps to precisely define and preserve concepts throughout the development process.**

***Keywords*—software architecture, architecture conformance checking, industrial study, ontologies**

I. Introduction

Software architecture analysis is an important means for quality control. One goal of architecture analysis is to assess that the system implementation adheres to its architecture. This process is also known as *architecture conformance checking* [1]–[3] and is an important activity of *architecture enforcement* [4]. The software architecture of a system is defined as *the set of principal design decisions about a system* [5]. Design decisions are not only comprised of project-specific architecture knowledge like architectural solution structures at different levels of abstraction, and the selection of implementation technologies, e.g., programming languages and frameworks, but also of project-generic knowledge, e.g., good software engineering principles like reference architectures and architectural styles. If the system implementation does not follow its intended architecture design, the system is at risk of not fulfilling its functional and non-functional requirements. The problem that the system implementation does not follow its defined architecture is known as architecture erosion [6], which contributes to architectural debt and may lead to problems during system maintenance and evolution.

Architecture erosion is a problem that occurs slowly over time. Therefore, architecture conformance analysis needs to be performed as a continuous process in order to support the early detection of problems. Continuous architecture conformance checking requires that this process is performed automatically, i.e., with a minimal demand of human resources. Automated conformance checking approaches require that architecture decisions are formalized in order to be automatically processed and verified against the source code.

Multiple approaches for automated architecture conformance checking have been proposed, e.g., [2], [3], [7] . These approaches provide a so-called *architectural concept language* for specifying architecture decisions. In this context, we refer to architecture decisions as architectural rules against which the implementation of a system can be checked. Existing approaches are typically restricted in terms of their provided architecture concept language and therefore only support a restricted set of architectural decisions that can be checked with a particular approach.

To overcome the limitation of existing approaches, we have developed an approach for architecture conformance checking that is based on ontologies. The approach supports the definition of flexible architecture concept languages to allow a formalization and automatic validation of architectural rules against an implementation [8], [9]. The formalization of ontology-based rules is ”hidden” behind a Controlled Natural Language (CNL) [10] that allows for the definition of architectural rules in a natural way. In this paper, we report on our experiences of applying our CNL/ontology-based approach in an industrial context. We conducted a case study in which we have analyzed if and to which degree architectural rules from three industrial projects can be formalized with our approach. The main contributions of our work are the following: (1) We have analyzed three industrial projects to find out which kinds of architectural rules exist in practice. (2) We have analyzed if these rules can be expressed an CNL/ontology-based approach. (3) We survey related work and discuss how our approach can be combined with existing approaches.

The remainder of this paper is organized as follows: Section II motivates the need for a novel approach and provides an overview on the conceptual background of the ontology-based conformance checking approach. In Section III we describe

2640-0715/19/$31.00 ©2019 IEEE
DOI 10.1109/APSEC48747.2019.00017

Authorized licensed use limited to: Bibliothekssystem Universitaet Hamburg. Downloaded on July 06,2020 at 08:47:24 UTC from IEEE Xplore. Restrictions apply.

how we have validated our work in an industrial case study. Section IV discusses the threats to validity. In Section V, we present related work and discuss to which extent related approaches are able to formalize the discovered architectural rules. Section VI concludes the paper with a summary of the main contributions and findings and provides an outlook on future research challenges.

II. Background

A. Challenges when Specifying Intended Software Architectures

Architecture Conformance Checking (ACC) approaches allow for the verification whether the *implemented architecture* adheres to the *intended architecture* [11]. ACC approaches require a description of the intended architecture of the software system as an input.

A lot of ACC tools are available where each tool provides a proprietary language to describe the intended architecture. This language defines so-called *architectural concepts* and *architectural relations*. For example, the tools Structure101[1] and Sonargraph[2] provide (graphical) languages to support the description of layered software architectures. They provide concepts like *layer* and the *use* relationships prescribing how layers are allowed to relate with each other. However, describing intended software architectures that do not follow a layered design becomes tedious. For example, in the context of an experience report by Fontana et al. [12], the intended architecture of the Ant tool[3] was described with Structure101. Due to the imposed specification language of Structure101, the authors were forced to describe the intended architecture of Ant in terms of the layers pattern. However, no evidence can be found in the documentation of Ant whether the layer pattern is actually implemented. As a result, the authors needed to explicitly state exceptions to the layer constraints, i.e., allowing bottom layers to access the upper ones. However, this way of specification is against the intention of the provided specification language. This example emphasizes the consequences of predefined language with fixed, not extendable semantics. This means, software architects 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 provides 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, reducing the maintainability and understandability of the architectural rule formalization.

Other authors have also observed this phenomenon, e.g., Woods [13] and Völter [14]. Dragomir et al. refer to this as the *meta-model incompatibility problem* [15]. That is why, in order to be sufficiently flexible, the language should be customizable with user-defined architecture concepts and relations. We have developed an approach that allows software architects to flexibly define their own language to specify the intended architecture in terms of architectural rules that can be verified against the source code.

B. Ontology-based Formalization of Architectural Rules

The main idea of the approach is to use ontologies for formalizing architectural rules. With ontologies, an application domain can be flexibly modeled using concepts, relations, and axioms. That is why, we first outline the basics of ontologies and description logics (DL). Then, we describe how ontologies can be used to formalize architectural rules. Additionally, we present our Controlled Natural Language (CNL) that serves as a frontend for architectural rule formalization.

Ontologies and Description Logics. Ontologies are used to represent the knowledge of a domain in a structured and formal way. For this, the domain is described in terms of *concepts*, *roles*, and *individuals*. *Concepts* (also called *classes*) represent sets of individuals characterized by common properties. *Roles* define binary associations (or relations) between *concepts* of a domain. *Individuals* describe concrete instances of concepts. Description logics (DLs) provide powerful languages to describe ontologies. DLs are a family of logic-based knowledge presentation formalism. Using DL, concepts can be flexibly defined with concept descriptions. Those are expressions built from atomic concepts and atomic roles using the constructors for concepts and roles provided by DL. Constructors used in our approach are the universal restriction ($\forall R.C$), the existential restriction ($\exists R.C$), the qualified number restrictions ($\geq nR.C$, $\leq nR.C$, $= nR.C$), the intersection ($C \sqcap D$), and the union ($C \sqcup D$)[4]. Concepts are defined by *general concept inclusions* (GCI) of the form $C \sqsubseteq D$. These are also called *class axioms*. Those axioms describe a *is-a* relationship of two concept descriptions C and D. This means, individuals of concept C must fulfill the properties defined by D in order to be consistent.
Ontologies (and DLs as their formal counterpart) are not restricted to any concepts and relations of a specific domain. That is why, any domain can be described and structured with ontologies. As we will show in the next sections, we exploit this flexibility for formalizing architectural concept language and architectural rules.

Ontology-based Specification of Architectural Rules. With our approach we allow software architects to flexibly define so-called architectural concept languages. Such a language defines architectural concepts used to describe

[1] https://structure101.com/
[2] https://www.hello2morrow.com/products/sonargraph
[3] https://ant.apache.org/

[4] C and D are atomic concepts or concept descriptions; R is an atomic role or a role description; n is a non-negative natural number.

Authorized licensed use limited to: Bibliothekssystem Universitaet Hamburg. Downloaded on July 06,2020 at 08:47:24 UTC from IEEE Xplore. Restrictions apply.

the software system on an architectural level as well as the architectural relationships between concepts. Architectural concepts describe architecturally-relevant core abstractions of the conceptual software architecture [14]. Concepts represent a set of concrete architectural entities with common properties and have a name. Architectural concepts are connected via named architectural relationships. For example, architectural concepts in component-based systems are *component* and *interface* that can be connected via the architectural relationship *provide* (a component provides an interface). Additionally, an architectural concept language contains a set of *architectural rules* that need to be satisfied by architectural concepts. *Architectural rules* describe responsibilities for architecture concepts in terms of permissions, prohibitions, and obligations, and how they are allowed to or not allowed to be related with each other via architectural relationships.

Ontologies and DLs can be used to capture the architectural concept language by mapping architectural concepts to concepts in DL and mapping architectural relations to roles correspondingly. As a consequence, architectural rules are formalized as class axioms. The DL constructors provide the necessary means to express rule semantics. Unfortunately, the DL formalism requires experts to formulate architectural rules as class axioms. Our approach therefore provides a so-called Controlled Natural Language (CNL) with which software architects and developers specify architectural rules for architectural concepts. A detailed description of the syntax and semantics of the CNL has already been given in [9]. In this section, the most important characteristics of the language will be described.

The CNL is a textual language defining only a small set of predefined keywords. Its syntax is very similar to natural language. This means, architectural rules are written as natural language sentences. However, the grammar is restricted in such a way that the meaning of sentences is unambiguous.

Using the CNL, an architectural rule can be written as follows:

```
Every Repository must use an Entity.
```

In this example, two architectural concepts are introduced, namely `Repository` and `Entity`. Those two concepts are connected via the architectural relation `use`. They keywords `Every`, `must`, and `an` are predefined by the CNL grammar. However, it is important to note that the concepts `Repository` and `Entity`, and the relation `use` are not predefined by the CNL. The user of the CNL is free to choose the vocabulary for concepts and relations and to define their concrete meaning by architectural rules. 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.

Formally, this rule is represented as a class axiom in DL. The modifier *must* defines the rule semantics. The rule semantics is given by the corresponding semantics of the constructor used to formalize the rule. In the example, `must` is mapped to the existential constructor of the DL formalism. In this rule, a relationship between instances of the two concepts must exist. Otherwise, the rule is violated. In our approach, we designed seven rule types based on the available DL constructors for formalizing architectural rules [9]. More specifically, we use the SROIQ description logic which is known to be sufficiently expressive with the advantage of being decidable [16].

III. Validation

In this section, we report how we have validated our approach for ontology-based architecture rule formalization.

A. Case Study Design

1) Objective and Research Questions: In this paper we validate the actual *flexibility* of our approach by applying it on architectural rules identified in industrial projects. Therefore we investigated the following research questions (RQs):

> **RQ1:** *What kind of architectural rules exist in industrial projects?*

With this question we investigate the different kinds of rules that exist in projects, which characteristics they have, and thus need to be supported by conformance checking approaches.

> **RQ2:** *In how far is the approach able to formalize those architectural 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 our ontology-based approach. We further analyze the characteristics of rules that cannot be formalized.

2) Units of Observation: We have analyzed our approach 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 thousands 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 domain-specific language, 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.

Authorized licensed use limited to: Bibliothekssystem Universitaet Hamburg. Downloaded on July 06,2020 at 08:47:24 UTC from IEEE Xplore. Restrictions apply.

TABLE I. 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

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, we have asked the software architects of the three projects (see above) to collect architectural rules in their projects. These rules were provided by the architects as part of the architecture documentation of theses projects. The rules are described (informally) in natural language (English). In order to answer **RQ1**, we have used artifact analysis as a technique for characterizing the architectural rules. We applied open coding and the constant comparison method [17]. Architectural 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, architectural rules provided by the software architects are formalized with the CNL notation of our approach by one of the authors. After this, all architects were interviewed. Each architect has been interviewed once. Each interview has been performed by the same interviewing author. In these interviews, the architects were asked to assess the CNL-based formalization of the architectural rules they have provided, i.e., each architect evaluates the formalization of the rules of his/her 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 CNL-based formalization. The rules have been processed successively. For each rule, the architects were asked by the interviewer to judge whether the CNL-based representation of the rules still reflects the intents of the original rules. This aims for answering **RQ2**, i.e., in how far the approach is able to formalize architectural rules found in practice. The

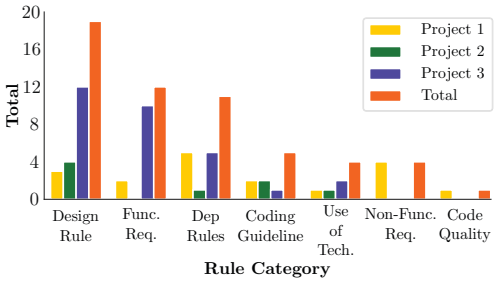

Fig. 1. Total number of rule categories found in the projects.

interview was performed following a interview guide with closed and open-ended questions according to the guidelines of [18]. The interview guide can be found in the supplementary[5]. It contains questions to capture the background of the participant, questions that aim to assess the suitability of the formalization of the architectural 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 I 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. One of the authors browsed the transcripts for passages relevant for research question. 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 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 [17]. With this method we reveal reasons for inappropriate formalization and other aspects.

B. RQ1: What kind of architectural rules exist in industrial projects?

In total, we found 56 architectural rules, where project 1 contains 18 architectural rules, project 2 contains 8 architectural rules, and project 3 contains 30 architectural rules. Hence, based on the analysis, we found the following types of architectural rules:

[5]https://tinyurl.com/y3k3jqd5

Authorized licensed use limited to: Bibliothekssystem Universitaet Hamburg. Downloaded on July 06,2020 at 08:47:24 UTC from IEEE Xplore. Restrictions apply.

- **Design rules**: Rules that enforce that system parts are realized in a prescribed way.
- **Functional requirements**: Rules that define specific program functionality.
- **Static Dependency Rules**: Rules defining how system parts are allowed/not allowed to interact with 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”.

Figure 1 shows how often a category was found in each project. Based on the codes and the resulting categories, we observed that the architectural 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 architectural rules that are defined on a high level of abstraction. This type of rules, for example, defines which architecture patterns must be used, enforces specific architectural 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 architectural rule documentation, especially in project 3. The third strongly represented category is the static dependency rules category. This rule category is well supported by existing approaches and tools such as [2], [3], [7].

Answer to RQ1: Based on the results, it can be observed that architectural rules with different characteristics exist in industrial projects. Moreover, we 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, we found that a significant proportion of the rule categories have been found that are not supported by these tools, e.g., **design rules** that make up **40%** of the architectural rules discovered.

C. RQ2: In how far is the approach able to formalize those architectural rules from practice?

We further analyzed the amount of rules that can be successfully formalized with our approach. We classify a rule as ”successfully formalized” when the corresponding software architect of the project approved that the original intention of the rule was appropriately reflected in the formalization during the interview.

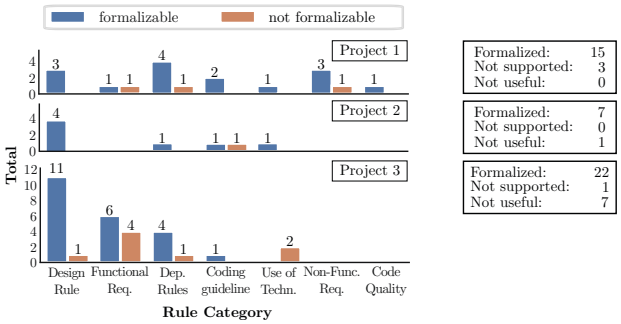

Fig. 2. Amount of rules in each project - categorized by the rule categories - that can be formalized and that cannot be formalized with our approach.

For each project, Figure 2 visualizes the amount of rules that can be formalized and that cannot be formalized with the ontology-based approach 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”).

The approach is **able to formalize 44 out of 56 (≈ 78.5%)** architectural rules from the projects. Table II depicts an excerpt of architectural rules taken from the industrial projects and their corresponding ontology-based formalization in the CNL notation. The rules can be formalized without loss of their original intention, since architectural concepts and relations used in the natural language description can be directly adopted in the formalization. Some rules in CNL even directly represent the natural language description, see for example rule 1 in Table II. The approach therefore provides a great flexibility in terms of architectural rule formalization.

However, it can be observed that **12 out of 56 (≈ 21.4 %)** architectural rules **cannot be formalized** with the approach. Some natural language descriptions contain constructs that cannot be easily transformed to an ontology-based formalization, since the underlying formalism does not support the constructs. For example, some rules contain temporal constraints, e.g., rule 5 in Table II. Although a formalization is provided, the rule is not properly formalized. As can be seen, the temporal restriction 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.

Additionally, as stated by the interview participants, it was not considered reasonable and necessary to formalize all of the architectural rules. Based on the qualitative analysis, we identified several reasons for this: **1)** Violating this rule would

Authorized licensed use limited to: Bibliothekssystem Universitaet Hamburg. Downloaded on July 06,2020 at 08:47:24 UTC from IEEE Xplore. Restrictions apply.

TABLE II. RULE EXAMPLES WRITTEN IN NATURAL LANGUAGE (**NL**) FOR EACH RULE CATEGORY FOUND IN THE PROJECTS AND THEIR CORRESPONDING FORMALIZATION USING THE CNL 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	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 RESTController 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

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), and **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 the CNL. 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.

D. Further Observations

In general, participants find that the approach is applicable for formalizing architectural rules. Based on the analysis of the transcripts, we found that participants appreciate the flexibility and expressiveness of the approach. This was also mentioned by an interview participant: *"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 architectural 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 the approach 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, participants mentioned that applying the approach helps to improve the quality of architectural rules, since it supports to clarify and clearly define the architecture concept language used in a project.

Authorized licensed use limited to: Bibliothekssystem Universitaet Hamburg. Downloaded on July 06,2020 at 08:47:24 UTC from IEEE Xplore. Restrictions apply.

During the interviews, the software architects sometimes perceived that the formalization did not appropriately reflect the original intention of the architectural 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 terms for architectural 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 original 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 the CNL formalization. As a result, the original intention was reflected more appropriately in both descriptions (CNL 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 architectural rule documentation. This means that the approach aids to unambiguously define architecture concepts and relations and the corresponding architectural rules. 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).

Answer to RQ2: We successfully formalized nearly **80%** of the architectural rules found in the industrial projects. This means that the approach is able to preserve the original intention of the majority of the architectural rules. Compared to existing approaches for (static) ACC, the approach is not restricted to static dependency rules and is able to formalize architectural rules from other categories. The proportion of rules that could not be formalized **(20%)** is acceptable, since this set contains mostly rules (**4 out of 56 (7%)**) which are considered to be not useful to be formalized.

IV. Threats to Validity

In this part, we discuss the threats to validity of our study.

Conclusion validity: To mitigate 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, we have analyzed projects from different domains. The projects also used different technology stacks. To address the history threat we have selected projects that were running for multiple years and therefore provided stable architectures and well established architectural rules.

Construct validity: To ensure validity of the constructs, we clearly defined the research questions and systematically selected methods for data collection and analysis to answer the research questions.

External validity: With our study, we do not claim external validity due to the number of analyzed industrial projects (3) which is too small to generalize our findings. This lets us conclude 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 are required by our approach, and how the language needs to be designed to better reflect the actual needs of its users. Nevertheless, our preliminary findings already provide valuable input for future research and have helped us identifying open research challenges.

V. Related Work

In this section, we compare related approaches and discuss to which extent they are capable of formalizing the rules found in the industrial projects.

The dependency constraint language (DCL) is a domain-specific language (DSL) allowing to specify module dependency rules [7]. However, this language is restricted in terms of the architecture concepts and relations. For example, it defines *module* as the only architecture concept. The DCL cannot be extended with other concepts. That is why, the approach is not suitable for formalizing most of the architectural rules found in the projects of our case study.

Dicto [2] uses a DSL to formalize architectural rules. It defines several rule types - similar to the types defined by our language. Each rule triggers a specific validation tool that is able to validate the rule type. With Dicto it is not possible to flexibly define architectural relations. Rules can only be described with predefined relations that are close to the code level, such as `depends on`, `have annotation`, or `calls`. Rules such as investigated in the projects (see Table II) can therefore not be formalized. For example, rule 1 requires a relation called `map-to`. This relation is not supported by Dicto. Assuming that HTTP error codes are implemented by Java Annotations[6], the rule could be formalized using the relation `have annotation`. However, this formalization would not represent the original intention of the rule.

Tools like Sonargraph[7] and HUSACCT [3] aim to formalize the intended architecture with graphical models. For example, Sotograph allows to specify the architecture using the concepts layers, slices, and subsystems. Since those tools are restricted to dependency analysis, it is not possible to formalize rules of other categories.

The approach proposed by Herold [1] is based on first-order logic. Since this formalism is more expressive than description

[6]as shown here: https://www.baeldung.com/spring-mvc-controller-custom-http-status-code

[7]https://www.hello2morrow.com/products/sonargraph

Authorized licensed use limited to: Bibliothekssystem Universitaet Hamburg. Downloaded on July 06,2020 at 08:47:24 UTC from IEEE Xplore. Restrictions apply.

logic - the formalism our approach is based on - it provides a very flexible approach for architectural rule formalization and validation. Our approach could be combined with such an expressive formalism in order to formalize rules that cannot be expressed with our approach. However, the approach by Herold does not provide a CNL in order to hide the formalism behind an understandable interface. It needs to be analyzed in how far our language can be adopted to represent this formalism.

Architecture description languages (ADLs) are another means to specify software architectures such as xADL 3.0 [19]. Our approach distinguishes from ADLs in two aspects: Firstly, ADLs provide very specific modeling solutions, e.g. with respect of the application domain or architectural views. The ontology-based approach aims for being applicable for a great range of domains and architectural aspects. Secondly, the primary purpose of ADLs is to provide a means for architectural modeling. Our approach aims for supporting architectural rule documentation and architecture conformance checking. Architecture conformance checking is typically not supported by ADLs.

Well-known software architecture documentation frameworks and templates such as [20] allow architects to capture relevant architectural information and views. Lightweight architecture decision documentation frameworks such as Architecture Decision Records (ADR)[8] or arc42[9] allow for the capturing of architecture decisions.

We strongly emphasize that we do not see our approach as a competitor to such approaches, i.e., ADLs and documentation frameworks, but rather to be complementary. Existing approaches could greatly benefit from the CNL as it enriches architecture documentation with verifiable architectural rules.

VI. Conclusion

In this paper, we have formalized architectural rules from three industrial projects with an ontology-based approach. We have shown that in practice a great variety of architectural rules with different characteristics exists. These different kinds of rules show that architecture conformance checking tools need to provide support for different kinds of architectural rules for comprehensive analysis.

The study results show that the majority of the discovered architectural rules could be successfully formalized. Only for a few rules, a formalization was considered to be unnecessary, e.g., because the rule is already enforced by the infrastructure of the system. It could be shown for the vast majority of useful rules, the formalization preserves the original intention. In many cases, the formalization even helped to refine the original rule, since the approach allows to define and use architectural concepts and relations more consistently. Consequently, even without verifying rules automatically, a project can benefit from such a precise architectural rule formalization. Additionally, existing approaches could greatly benefit from including our approach, because it would add the necessary flexibility to properly reflect the respective project language used to describe the software architecture.

[8]http://thinkrelevance.com/blog/2011/11/15/documenting-architecture-decisions

[9]https://docs.arc42.org/section-9/

VII. Acknowledgments

The research reported in this paper has been partly supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry for Digital and Economic Affairs, and the Province of Upper Austria in the frame of the COMET center SCCH.

References

[1] S. Herold, "Architectural compliance in component-based systems" Ph.D. dissertation, Clausthal University of Technology, 2011.

[2] A. Caracciolo, M. F. Lungu, and O. Nierstrasz, "A unified approach to architecture conformance checking," in Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture, Montreal, 2015, pp. 41-50.

[3] L. J. Pruijt, C. Köppe, J. M. van derWerf, and S. 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, Vasteras, Sweden, 2014, pp. 851–854.

[4] S. Schröder, M. Soliman, and M. Riebisch "Architecture enforcement concerns and activities - an expert study," in Journal of Systems and Software, vol. 145, 2018, pp. 79–97.

[5] R. Taylor, N. Medvidovic, and E. Dashofy, Software Architecture: Foundations, Theory, and Practice. Wiley, 2009.

[6] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey" in Journal of Systems and Software, 2012, pp. 132-151.

[7] R. Terra and M. T. Valente, "A dependency constraint language to manage object-oriented software architectures" in Software: Practice and Experience, vol. 39, 2009, pp. 1073-1094.

[8] S. Schröder and M. Riebisch, "Architecture conformance checking with description logics," in Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings, Canterbury, 2017, pp. 166–172.

[9] S. Schröder and M. Riebisch, "An ontology-based approach for documenting and validating architecture rules", in Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings, Madrid, 2018, pp. 52:1-52:7.

[10] R. Schwitter, "Controlled natural languages for knowledge representation," in Proceedings of the 23rd International Conference on Computational Linguistics: Posters, Beijing, 2010, pp. 1113-1121.

[11] J. Knodel and M. Naab, Pragmatic Evaluation of Software Architectures, 1st ed. Springer Publishing Company, 2016.

[12] F. A. Fontana, R. Roveda, M. Zanoni, C. Raibulet, and R. Capilla, "An experience report on detecting and repairing software architecture erosion," in the Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture, Venice, 2016, pp. 21–30.

[13] E. Woods and R. Hilliard, "Architecture description languages in practice session report," in the Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, Pittsburgh, 2005, pp. 243–246.

[14] M. Völter, "Architecture as language," in IEEE Software, vol. 27, 2010, pp. 56-64.

[15] A. Nicolaescu, "Behavior-based architecture conformance checking," Ph.D. dissertation, RWTH Aachen University, Germany, 2018.

[16] F. Baader, The description logic handbook: Theory, implementation and applications, Cambridge University Press, 2003.

[17] B. G. Glaser, Theoretical sensitivity: Advances in the methodology of grounded theory, Sociology Press, 1978.

[18] K. Charmaz, Constructing grounded theory, 2nd ed., SAGE Publications, 2014.

[19] E. M. Dashofy, A. Van der Hoek, and R. N. Taylor, "A highly-extensible, xml-based architecture description language," in the Proceedings of the 2nd Working IEEE/IFIP Conference on Software Architecture, Amsterdam, 2001, pp. 103-112.

[20] D. Garlan, F. Bachmann, J. Ivers, J. Stafford, L. Bass, P. Clements, and P. Merson, Documenting Software Architectures: Views and Beyond, 2nd ed. Addison-Wesley Professional, 2010.

Authorized licensed use limited to: Bibliothekssystem Universitaet Hamburg. Downloaded on July 06,2020 at 08:47:24 UTC from IEEE Xplore. Restrictions apply.