An Ontology-Based Approach for Documenting and Validating Architecture Rules

Sandra Schröder and Matthias Riebisch
Universität Hamburg, Department Informatics
Hamburg, Germany
{schroeder,riebisch}@informatik.uni-hamburg.de

ABSTRACT

Architecture conformance checking is an important activity of architecture enforcement where the architect ensures that all architecture concepts are implemented correctly in the source code. In order to support the architect, a lot of tools for conformance checking are available that allow to formalize the architecture in order to perform an automated verification. Typically, the formalization uses a rigid, tool-specific architecture concept language that may strongly deviate from the project-specific architecture concept language. In addition, a high level of formal expertise is required in order to comprehend the created formalization. We present an approach that uses a controlled natural language for the formalization of architecture concepts. This language allows to flexibly express architecture rules directly with project-specific concepts. Consequently, the resulting formalization is easy to understand and might also be used as an architecture documentation at the same time. Nevertheless, the documentation can be automatically verified, since the approach is based on powerful means of the semantic web, i.e., ontologies and description logics. For the evaluation of the approach, we use the real-world software system TEAMMATES and show that architecture rules and concepts can be flexibly designed and checked for conformance in order to detect crucial architecture violations.

KEYWORDS

architecture erosion, description logics, ontologies, architecture documentation, architecture conformance checking

ACM Reference Format:
Sandra Schröder and Matthias Riebisch. 2018. An Ontology-Based Approach for Documenting and Validating Architecture Rules. In *12th European Conference on Software Architecture: Companion Proceedings (ECSA '18), September 24–28, 2018, Madrid, Spain.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3241403.3241457

1 INTRODUCTION

Architecture enforcement is a challenging responsibility of the software architect [12]. During enforcement, the software architect is concerned with ensuring that architecture design decisions are implemented correctly in the source code. He performs architecture conformance checking in order to reveal inconsistencies between the implementation and the intended software architecture [5]. For this, the software architect defines so-called *architecture rules* [3] that should continually be checked against the source code in order to detect *architecture violations* [16] and to eventually minimize architecture erosion [9].
A lot of tools and approaches have been developed, in order to support rule-based conformance checking, e.g. [3], [10], [16], or [4]. Each of them provide a so-called *architecture concept language* the intended software architecture is described with. However, we identify two issues that arise when using those tools:
I1: Lacking Flexibility of Architecture Concept Languages. Tools typically use a language that is fixed in terms of the architecture concepts the intended architecture and the architecture rules can be described with. That is why, the architect cannot extend the language by new concepts that are needed in the project. He is forced to reformulate the project-specific architecture concepts using the available architecture concepts of the tool, e.g. *module*, *component*, or *layer*. Consequently, the original intentions of the architecture rules are prone to get lost during the formalization.
I2: Inappropriate Architecture Rule Documentation. As mentioned in [1] and [8] there is still a lack of approaches that provide usable and readable architecture formalizations. Moreover, those approaches typically do not integrate well in the toolchain used by developers, e.g. such as plain text documentation in asciidoc or markdown format. Therefore, architecture rules cannot be appropriately preserved and documented. In case architecture rules are documented, only natural language descriptions are used or formal descriptions are supported with natural language explanations. 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.

We propose an ontology-based approach in order to address the issues described above. An ontology defines a vocabulary that describes concepts and relations that are representational for a domain. In the context of architecture conformance checking, an ontology provides a flexible architecture concept language with which the software architect can describe the software architecture and corresponding architecture rules. 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. Additionally, description logics [2] are the formal basis for ontology languages. That is why, mature and efficient reasoning services can be exploited in order to check the architecture concept language for inconsistencies and to validate whether architecture rules are violated. This addresses issue I1. In order to address I2, we design a *controlled natural language* (CNL). CNLs are a subset

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ECSA '18, September 24–28, 2018, Madrid, Spain
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6483-6/18/09.
https://doi.org/10.1145/3241403.3241457

of natural languages that use a restricted vocabulary and grammar [7]. They aim to bridge the gap between natural and formal languages by increasing understandability and unambiguity. As they are very similar to natural languages and use a well-defined syntax and semantics, they are understandable for humans and can be processed by machines at the same time. CNLs integrate well with the description logic formalism and ontologies, so that they are frequently applied as knowledge representation languages, as in [13]. However, to the best of our knowledge, there are no approaches that apply CNLs for architecture rule formalization. With the CNL, 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. The CNL can simply be integrated in plain text files that can additionally be stored in version control systems along with the source code. Therefore, our approach facilitates an *readable and verifiable architecture documentation.*
The contributions of this paper are twofold: Firstly, we develop an approach based on description logics and ontologies that supports a more flexible way to define architecture concepts and relations specific to the respective project (Section 3). Secondly, we design a CNL that enables a more comprehensible and readable formalization of architecture rules, and that also provides a basis for architecture documentation (Section 3.2). We evaluate our approach using a real-world software system and show how its architecture rules can be flexibly and unambiguously formalized with the CNL. We additionally show that the approach is suitable to detect crucial architecture violations (Section 4).

2 BACKGROUND ON DESCRIPTION LOGICS, ONTOLOGIES, AND SEMANTIC WEB TECHNOLOGIES

Description logics (DL) are a family of logic-based knowledge-presentation formalisms [2]. They are used to represent the knowledge of a domain in a structured and formal way. In DL, the domain is formalized using *concepts*, *roles*, and *individuals* based on a formally defined syntax and semantics. Concepts 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. Concepts can be flexibly defined using concept descriptions. Those are expressions built from atomic concepts and atomic roles using the constructors for concepts and roles provided by description logics. Those constructors are for example 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 ($C \sqcup D$)[1].
In addition, there are *general concept inclusions* (GCI) of the form $C \sqsubseteq D$. These are axioms describing a *is-a* relationship of two concept descriptions C and D, where individuals of C must satisfy the properties of D in order to be consistent. For example, we can describe that an architectural component needs to provide at least two interfaces with the GCI $Component \sqsubseteq \geq 2 provide.Interface$. Those types of axioms are contained in the so-called terminological box (TBox). As we will see later in Section 3 and 4, we use the TBox to describe architecture concepts and relations and especially use GCIs in order to formalize architecture rules based on those concepts and relations. We can further assert facts about concrete instances of the concepts contained in the TBox. Those facts are part of the assertional box (ABox). ABox axioms capture knowledge about named individuals by stating to which concepts they belong (i.e. concept assertions) and how they are related with each other (i.e. role assertions). With respect to the above example, we can add the statements $Component(c1)$, $Interface(i1)$, $Interface(i2)$, $provide(c1, i1)$, $provide(c1, i2)$ to the ABox in order to describe that there exists one concrete component (the named individual c1) and two interfaces (individuals i1 and i2) that are provided by this component. Together, the TBox and the ABox constitute a so-called *knowledge base* [2, 6].

[1] 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.

3 FLEXIBLE ARCHITECTURE CONFORMANCE CHECKING

In order to check the implementation for violations, the following inputs are necessary [11]:

(1) architecture concepts, relations, and rules that together constitute the architecture concept language,
(2) the source code represented as an ontology (e.g. Java source code),
(3) an architecture-to-code-mapping in order to identify architecture concepts in the source code.

The architecture concept language and the architecture-to-code-mapping are provided by the software architect, whereas the source code is automatically transformed into an ontology. The software architect defines a project-specific architecture ontology using the CNL introduced later. Architecture concepts map to atomic concepts of the description logic formalism, whereas relations between concepts are modeled as atomic roles. Moreover, the architect defines architecture rules based on the concepts. The CNL statements are automatically converted into *OWL integrity constraints* [15]. Those are then integrated into the knowledge base (Stardog[2]) that also contains the source code ontology. The architecture concepts are automatically extracted from the code driven by the architecture-to-code-mapping that was provided by the architect. Thus, the implemented architecture is extracted. The mapping can be implemented in arbitrary ways in order to detect architecture concepts in source code. In our case, we apply SWRL rules in order to describe mapping rules. However, it is also possible to apply machine learning algorithms or other approaches in order to detect architecture concepts in the code. Subsequently, the implemented architecture is evaluated against the architecture rules using reasoning. Inconsistencies detected in the knowledge base correspond to architecture violations.

3.1 Specifying Architecture Rules as OWL Integrity Constraints

As mentioned before, we use OWL integrity constraints - that correspond to architecture rule types - to formalize architecture

[2] https://www.stardog.com/

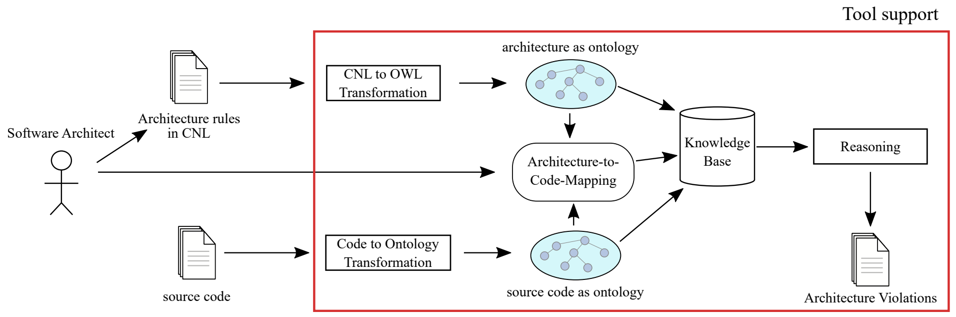


Figure 1: Overview of the conformance checking approach.

rules. In contrast to general OWL class axioms, integrity constraints realize the so-called *closed-world assumption* [15] that assumes the ABox to be completely specified. Consequently, missing assertions are able to create violations. For example, the rule $Repository \sqsubseteq \exists manage.Entity$ ("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 the following, we propose seven types of integrity constraints that are fundamental for supporting architecture rule formalization. Note that the given rule types can be easily extended with new ones, as the description logic formalism supports a flexible definition of concepts by using concept constructors. The semantics of each rule type is defined by the semantics of description logics using the closed-world assumption. Table 1 depicts the rule types in their description logic representation and an example for each rule type. The rule types have the following meaning:

(1) *Sub-concept Rule type:* An individual of an architecture concept must also satisfy the properties of its parent architecture concept.
(2) *Domain-Range Rule type:* An individual can only belong to a specific architecture concept in order to have a relation to an individual that belongs to another architecture concept.
(3) *Existential Rule type:* Every individual of a specific concept must have a specific relation with an individual of another architecture concept.
(4) *Universal Rule type:* Every individual of a specific concept can only have a specific relation with an individual of a specific architecture concept.
(5) *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 non-negative natural number.
(6) *Conditional Rule type:* An individual of a specific concept can only have a relation with an individual of another architecture concept, if there already exists another type of relation between them that is specified in the rule.
(7) *Negation Rule type:* An individual of a specific concept is not allowed to be related with an individual of another architecture concept.

To the best of our knowledge, this set of types of architecture rules covers more rule examples than all other existing approaches combined. For example, the DCL language [16] only implements the Domain-Range Rule type, the Negation Rule type, and the Existential Rule type. Dicto [3] additionally supports the Universal Rule type. The Sub-concept, Cardinality, and Conditional Rule types are not supported by those approaches. However, they are necessary to formalize common architecture rules as we will show in the evaluation (Section 4). In the subsequent part of this section, we illustrate how a CNL can be used to express the above rule types in a more comprehensible way.

```
⟨sentence⟩ ::= (⟨subject⟩ [‘must’|‘can’] ⟨roleExpression⟩ ⟨object⟩‘.’)
   |  (‘If’ ⟨conceptID⟩ ⟨role⟩ ‘a’ ⟨object⟩ [‘then’ | ‘,’] ‘it’ ‘must’ ⟨role⟩ ‘this’
        ⟨object⟩‘.’)

⟨subject⟩ ::= ‘No’ ⟨conceptID⟩ | (‘Every’|‘Each’)? ⟨conceptID⟩

⟨roleExpression⟩ ::= ‘only’ ⟨role⟩ (‘a’|‘an’)?
   |  ‘be’ (‘a’|‘an’)
   |  ⟨role⟩ (‘at-most’|‘at-least’|‘exactly’) ⟨count⟩
   |  ⟨role⟩ (‘a’|‘an’)?

⟨object⟩ ::= ⟨conceptID⟩ (‘or’ ⟨conceptID⟩)*

⟨conceptID⟩ ::= ((’a’..’z’)|(’A’..’Z’))+((’A’..’Z’)|(’a’..’z’))*
```

Grammar 1: Grammar of the controlled natural language in EBNF.

3.2 Controlled Natural Language

We designed a CNL that aims to provide a more readable and understandable architecture specification. The goal of the design of the CNL is to support at least the classified rule types depicted in Table 1. That is why, we need to define a suitable grammar that prescribes the sentence structure and keywords that allow for the construction of the sentences. We base the design of our CNL on existing ones that aim to integrate a natural language layer for ontology authoring such as Rabbit or ACE [14]. In contrast to those languages, we also support keywords that reflect the closed-world assumption, e.g. by adding modal words like `must` or `can`. Grammar 1 depicts the grammar of the CNL in EBNF notation. The vocabulary consists of content words that describe concepts and roles, and of predefined keywords. The content words represent names for architecture concepts (*conceptID* in Grammar 1) and relations (*role* in Grammar 1) that can be arbitrarily chosen by the architect. Each concept identifier is represented as a noun that can be written as a sequence of words in camel case notation, whereas each role is represented by a verb.

Table 1: Supported rule types of the formalism with exemplary rules in their DL and CNL representation. A and B are atomic concepts or concept description, R is an atomic role, and n is a non-negative natural number.

Rule Type	Description Logic	CNL expression	Example
Sub-concept Rule	$A \sqsubseteq B$	`(Each/Every)` A `must be a/an` B.	Every AggregateRoot must be an entity.
Domain-Range Rule	$\exists R.\top \sqsubseteq A$, $\top \sqsubseteq \forall R.B$	`Only a/an` A `can` R `a/an` B.	Only a ServiceComponent can use a DAOs.
Existential Rule	$A \sqsubseteq \exists R.B$	`(Each/Every)` A `must` R `a/an` B.	Every Repository must manage an Entity.
Universal Rule	$A \sqsubseteq \forall R.B$	`(Each/Every)` A `can only` R `a/an` B	Every LogicType can only access a StorageApi.
Cardinality Rule	$A \sqsubseteq= nR.B$ $A \sqsubseteq\leq nR.B$ $A \sqsubseteq\geq nR.B$	`(Each/Every)` A R `exactly/at-most/at-least n` B.	Every Host contains (exactly, at most, at least) two ServiceInstances.
Conditional Rule	$R \sqsubseteq S$	`If` A R `a/an` B, `then it must` S `this` B.	If a LogicType uses a DBType, it must manage this DBType.
Negation Rule	$A \sqsubseteq \neg(\exists R.B)$	`No` A `can` R `a/an` B.	No DAO can use a BusinessLogicComponent.

The CNL uses only a small set of predefined keywords reserved for special purposes. For example, the keywords `Each, Every, No,` and `If` are necessary to start a sentence. `Each` and `Every` can be used interchangeably and are optional. For example, the sentences `Every AggregateRoot must be an Entity`, `Each AggregateRoot must be an Entity`, and `AggregateRoot must be an Entity` are equivalent. The keyword `No` introduces a negation rule. Using the keyword `If` starts a conditional sentence. We use the keywords `at-most`, `at-least`, and `exactly` combined with a non-negative natural number in order to allow the construction of cardinality rules. We also support modal words like `must` and `can` that are necessary for writing a sub-concept, domain-range, existential, conditional or a universal rule. There is a bijective mapping between the rule types and the CNL sentences that can be constructed according to the grammar. Each rule type from Table 1 can be written as a CNL sentence and each CNL sentence can be mapped to a rule type. Therefore, the semantics of the CNL is given by the semantics of the respective rule type. The rule types are constructed in CNL as depicted in Table 1. Using the CNL, the architecture ontology and the corresponding rules are generated automatically. The ontology is stored as an OWL file that is eventually imported into the knowledge base, so that architecture conformance checking can be performed.
Note that no additional statements for defining the architecture concepts and relations are necessary. While writing architecture rules as CNL 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 DL 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 CNL sentence is transformed into an OWL class axiom that connects the classes and relation with each other.

3.3 Tool Support

A prototype for tool support has been implemented. Architecture rules can be documented inside a plain text file. Currently, they can be integrated in a text file that follows the asciidoc format[3]. This has the advantage that the rules can be easily managed in a version control system. Moreover, a lot of asciidoc and markdown templates for architecture documentation exist, so that architecture documentations can be enriched with architecture rule formalizations.
The architecture rule documentation is automatically converted into integrity constraints. The tool prototype is also able to automatically transform Java source code into the code ontology. Based on the mapping rules specified by the architects or developers, the implemented architecture is extracted by using a reasoner provided by Apache Jena. The implemented architecture is imported to the knowledge base (Stardog Knowledge Graph Platform) 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.

4 CASE STUDY

In the following, we present the evaluation of our approach. We apply our approach to the open source project *TEAMMATES*[4], a web-based feedback management tool for education. We have chosen this software system for evaluation purposes as it provides an up-to-date comprehensive documentation of the architecture design. In particular, the architecture rules are available. The goal of this evaluation is to show a) that the approach finds crucial architecture violations and b) that the approach allows a flexible architecture rule definition. In order to evaluate the quality of the violation detection of our approach, we use the detection results[5] of the tool HUSACCT [10] that was applied during the Saerocon Workshop[6]. The results are used as a ground truth in order to compare them with the architecture violations found with our approach. Additionally, the violations were prioritized by the TEAMMATES

[3]http://asciidoc.org/
[4]https://github.com/TEAMMATES/teammates
[5]https://github.com/sebastianherold/SAEroConRepo/wiki/teammates-5.110-code-mapping-husacct
[6]https://saerocon.wordpress.com/

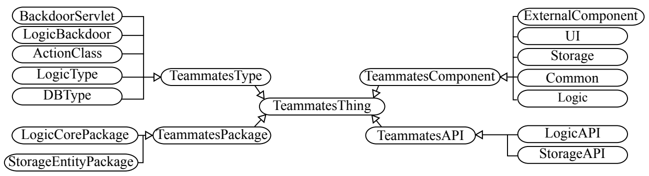


Figure 2: Architecture ontology of TEAMMATES .

architect. We use this prioritization in order to evaluate whether our approach finds crucial violations. In order to show the flexibility, we tailor an architecture concept language by defining architecture concepts and relations as they are needed to specifically describe the architecture of TEAMMATES and to formalize the corresponding architecture rules. The TEAMMATES architecture defines architecture rules typically found in architecture documentations [3]. We show that the rule types from Section 3 are suitable to formalize all architecture rules of TEAMMATES.

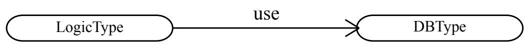
4.1 Defining the Architecture Concept Language and Architecture Rules of TEAMMATES

The TEAMMATES developers documented all the important architecture rules that need to be followed by the implementation. We use the documentation in order to formalize the rules with our approach and check them against the ontology-based representation of the source code. In the following, we demonstrate how the architecture rules are formalized using our approach by a) defining an ontology of the main architecture concepts of TEAMMATES (see Figure 2), b) formalizing the architecture rules based on the concepts (see Table 2), and c) formalizing the architecture-to-code-mapping between the architecture and the code concepts using SWRL rules [7] so that we are able to perform the architecture conformance checking using reasoning services (see Figure 3). The ontology, the mapping rules, the formalized architecture rules, and the ontology of the source code of TEAMMATES are available as supplementary material[8].

4.1.1 Architecture Ontology and Rules of TEAMMATES. In order to create the architecture ontology, we investigated the architecture diagrams and documents provided in the repository of TEAMMATES. The analysis lead us to the categorization of the architecture concepts as presented in Figure 2. The leaf nodes are the concepts as they are used in the architecture documentation of TEAMMATES. For those concepts, architecture rules will be defined. Note that this is a pure analysis of the architecture concepts, where its result does not directly constitute to the knowledge base. The architecture ontology is indirectly built by formalizing the architecture rules for the concepts with the CNL. After having identified the main architecture concepts, the architecture rules are defined subsequently based on those concepts. Selected architecture rules and their corresponding formalization in DL and CNL are listed in Table 2. For the selected subset in Table 2 we applied the rule types Conditional Rule, Universal Rule, and Domain-Range Rule.

[7] https: //www.w3.org/Submission/SWRL/

[8] https://swk-www.informatik.uni-hamburg.de/~saerocon2018-supplementary/



(A) *LogicType* Mapping

```
[isLogicType: (?class rdf:type code:JavaClass)(?class code:hasName ?name)
regex(?name, '\\w*Logic') -> (?class rdf:type architecture:LogicType)]
```

(B) *use* Mapping

```
[useMapping: (?class rdf:type code:JavaClass)
(?class rdf:type architecture:LogicType)
(?classDB rdf:type code:JavaClass)(?classDB rdf:type architecture:DBType)
(?class code:dependsOn ?classDB)->(?class architecture:use ?classDB)]
```

Figure 3: Mapping the architecture concepts *LogicType* and the architecture-level *use* relationship to Java code concepts.

The remaining rules are given in the supplementary information on our website(formalized in CNL and the generated OWL file).

4.1.2 Architecture-to-Code-Mapping. The architecture-to-code-mapping can be formalized based on several conventions, like class or package names, meta data (e.g. java annotations), or package structuring conventions. The developers of TEAMMATES use naming conventions in order to identify architecture concepts in the source code. We show two mapping rules for the concept *LogicType* and the architecture-level *use* relationship in Figure 3 that are mapped by such a naming convention. The rules express the following mappings:

- **(A)**, *LogicType* Mapping: If something is a *JavaClass* and has a name containing "Logic" as its suffix, then this class is identified as a *LogicType*. For the *DBType*, the same mapping is defined, but it applies the suffix "Db".
- **(B)**, *use* Mapping: If a Java class is a *LogicType* and another class is a *DBType* and there is a code-level *dependsOn* relationship (defined in the Java code ontology) from a class implementing a *LogicType* to another class that implements a *DBType*, then the *LogicType* uses this *DBType* via the architecture-level relationship *use*.

Each *LogicType* has a corresponding *DBType*. The developers use 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 architecture-level *manage* relationship between *LogicType* and *DBType* individuals.
Having defined the architecture concepts, rules, and the mapping, we import them to the knowledge base and applied the available reasoning services. First, the mapping is executed in order to extract the implemented architecture. Then, the reasoning services are used to find inconsistencies, i.e. architecture violations. The conformance checking results for the rules in Table 2 are presented in the following.

4.2 Architecture Conformance Checking Results

We checked the architecture rules 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. We found violations of the rules depicted in Table 2. All relevant violations have been found. Some of them, especially the

Table 2: A subset of the architecture rules of TEAMMATES specified in different notations, namely natural language (NL), in Description Logic (DL), and in Controlled Natural Language (CNL).

R1	NL	Each LogicType can only use its corresponding DBType.
	DL	$use \sqsubseteq manage$
	CNL	`If LogicType uses a DBType, it must manage this DBType.`
R2	NL	Logic classes can only access classes from the storage API.
	DL	$LogicType \sqsubseteq \forall access.(StorageAPI \sqcup Common \sqcup ExternalComponent \sqcup Logic)$
	CNL	`Logic can only access StorageAPI or Common or ExternalComponent or Logic.`
R3	NL	Common should not have access to any packages, except `storage::entity`.
	DL	$Common \sqsubseteq \forall access.(Common \sqcup StorageEntityPackage)$
	CNL	`Common can only access Common or StorageEntityPackage.`
R4	NL	Only *Action can touch logic api.
	DL	$\exists touch.\top \sqsubseteq ActionClass, \top \sqsubseteq \forall touch.LogicAPI$
	CNL	`Only ActionClass can touch LogicAPI.`

dependency between the classes `FeedbackResponsesLogic` and `FeedbackResponse` (implementing the *StorageEntityPackage* concept) have been described as *"severe and should be fixed"* by the architect of TEAMMATES during the workshop discussion. **R1** is violated ten times by the class `BackDoorLogic`, since it uses classes that implement the *DBType* concept but it does not manage those *DBType*s (`BackDoorDb` does not exist). Those are real violations found by our approach that cannot be found with the HUSACCT formalism, since it simply does not support the conditional rule type (see Table 1). Consequently, the violations have to be regarded true violations. For rule **R4** we also found several violations 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. Consequently, we regard the violations as true violations. In summary, the approach can be used to detect crucial architecture violations. Note that the detection quality greatly depends on how the mapping and the architecture rules are defined. This is a common challenge in architecture conformance checking.

4.3 Discussion

Flexibility of the architecture concept language. As illustrated in the paragraphs before, the approach allows us to define different types of architecture concepts and relations. Therefore, it allows us to flexibly define the architecture concept language as needed for rule formalization. For example, we are not restricted to concepts like *module* or *component*. Other concepts like *TeammatesType* can be defined that work on a lower level of abstraction of architecture that, nevertheless, entail crucial architecture rules. The architecture concept language can be extended with new concepts as needed, e.g. new sub-concepts for *TeammatesType*. This is not possible in approaches like [16]. We think that those approaches could greatly benefit from such an ontology-based approach.

Reducing ambiguities. The TEAMMATES developers formalize the architecture rules in an XML file[9] that is checked with the Macker tool[10]. In addition, each rule is supported by an informal, natural language description in order to summarize the XML-based formalization. We found that the XML and natural language description are not consistent in every case. For example, the natural language description of rule **R2** only allows architecture-level *access* relationships to the *StorageAPI*. By this, it does not allow such relationships to the *Common* component, the *ExternalComponent*, and to the *Logic* component itself. By further investigating the XML formalization of the rule, we found that this is not the actual intention of the architecture rule. This means, the natural language description is inconsistent with respect to the corresponding XML formalization used in the architecture documentation (see Section 1, issue I2). Note that our rule formalization in CNL accounts for the real intention of **R2** and therefore deviates from the natural language specification as given in the architecture documentation of TEAMMATES (see Table 2). Without explicitly allowing the *access* relationships to *Common*, *ExternalComponent*, and *Logic*, we would have found 99 more violations of rule **R2**. After refining the rule according to its real intention the number of violations of rule **R2** is reduced from 99 to one real violation, namely the dependency between the classes `FeedbackResponsesLogic` and `FeedbackResponse`. Another ambiguity can be observed for rule **R4**. In contrast to the natural language specification, the XML formalization also allows several other classes than the *ActionClass*es (e.g. the servlet classes) to touch the *LogicAPI*. Both aspects show, how our approach mitigates the risk of ambiguous and inconsistent architecture rule specifications, since the formalization is directly given by the CNL description.

Formalization effort and maintenance of rules. We are aware of challenges regarding the ontology-based approach. First, a certain amount of effort is needed in order to create the ontologies and the architecture-to-code-mapping. However, we think that 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. 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. Secondly, 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

[9]https://github.com/TEAMMATES/teammates/blob/master/static-analysis/teammates-macker.xml

[10]https://innig.net/macker/

add new architecture rules to be checked. It is necessary to make sure no concepts are redundantly introduced that already exist in the concept language with a different name. Decisions about new concepts and rules need to be thoroughly discussed within the team in order to mitigate this risk.

5 RELATED WORK

Several approaches and tools for architecture conformance checking have been developed. For example, the dependency constraint language is a domain-specific language (DSL) allowing to specify module dependency rules [16]. However, this language is restricted in terms of the architecture concepts and relations that can be used to define the intended architecture. It solely uses the architecture concept *module*. Other concepts cannot be added.

Dicto [3] also uses a DSL to formalize architecture rules. The provided DSL is a wrapper language where each architecture rule triggers a specific checking tool that validates the specific rule. In order to be able to support more rule types to be checked, other tools are needed and a wrapper needs to be implemented for this tool. This makes the approach less flexible in terms of new architecture concept definition. Additionally, there is no uniform representation of architecture and code (and the mapping between both) as we provide it by using ontologies.

The TEAMMATES developers use Macker for architecture conformance checking. This tool uses an XML representation of architecture rules. Such a rule specification can be very complex. Therefore, it is supported by a natural language description. In our approach, there is no need for additional natural language descriptions as the architecture rule formalization is self-explanatory. In addition, there is no guarantee that the natural language description is consistent with the actual formalization. This risk is mitigated in our approach as discussed in Section 4.

Source code query languages such as Semmle QL[11] allow for complex queries over the source code structure. Those languages can also be used for architecture conformance checking. However, by definition, architecture concepts and rules can only be modeled implicitly using queries.

Tools like Sonargraph [12], Lattix [13], Structure101[14], and HUSACCT [10] aim to formalize the intended architecture with graphical models. For example, Sonargraph allows to specify the architecture using the concepts layers, slices, and subsystems. Although the graphical representation helps to comprehend the intended architecture, this graphical language can not be extended with additional architecture concepts, relations, and rule types.

6 CONCLUSION AND FUTURE WORK

We presented an ontology-based approach for formalizing and verifying architecture rules that is based on ontologies, description logics, and a controlled natural language. Due to the flexibility of ontologies, the architect can freely define his own architecture concept language and is not bound to use the concept language provided by a specific tool. The controlled natural language supports the readability of the architecture rules. Due to the fact that it is also verifiable, there is no need to support the formalization with natural language descriptions that could be inconsistent with the actual formalization. That is why, our approach greatly supports living and self-validating architecture documentation. As shown in the evaluation, architecture rules can be automatically verified by exploiting the advantages of mature reasoning services based on description logics in order to reveal crucial architecture violations. In the future, we plan to conduct an expert study with software architects from industry in order to evaluate how they perceive the usefulness and usability of the CNL.

[11] https://semmle.com/products/semmle-ql/
[12] https://www.hello2morrow.com/products/sonargraph
[13] http://lattix.com/
[14] http://structure101.com/

REFERENCES

[1] Nour Ali, Sean Baker, Ross O'Crowley, Sebastian Herold, and Jim Buckley. 2017. Architecture consistency: State of the practice, challenges and requirements. *Empirical Software Engineering* (15 May 2017). https://doi.org/10.1007/s10664-017-9515-3

[2] Franz Baader. 2003. *The description logic handbook: Theory, implementation and applications.* Cambridge university press.

[3] A. Caracciolo, M. F. Lungu, and O. Nierstrasz. 2015. A Unified Approach to Architecture Conformance Checking. In *2015 12th Working IEEE/IFIP Conference on Software Architecture.* 41–50. https://doi.org/10.1109/WICSA.2015.11

[4] S. Herold, M. Mair, A. Rausch, and I. Schindler. 2013. Checking Conformance with Reference Architectures: A Case Study. In *2013 17th IEEE International Enterprise Distributed Object Computing Conference.* 71–80. https://doi.org/10.1109/EDOC.2013.17

[5] Jens Knodel and Matthias Naab. 2016. *Pragmatic Evaluation of Software Architectures* (1st ed.). Springer Publishing Company, Incorporated.

[6] Markus Krötzsch, Frantisek Simancik, and Ian Horrocks. 2012. A Description Logic Primer. *CoRR* abs/1201.4089 (2012). arXiv:1201.4089 http://arxiv.org/abs/1201.4089

[7] Tobias Kuhn. 2014. A Survey and Classification of Controlled Natural Languages. *Computational Linguistics* 40, 1 (2014), 121–170. https://doi.org/10.1162/COLI_a_00168

[8] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang. 2013. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering* 39, 6 (June 2013), 869–891. https://doi.org/10.1109/TSE.2012.74

[9] Dewayne E. Perry and Alexander L. Wolf. 1992. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes* 17, 4 (Oct. 1992), 40–52.

[10] Leo J. Pruijt, Christian Köppe, Jan Martijn van der Werf, and Sjaak Brinkkemper. 2014. 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).* ACM, New York, NY, USA, 851–854. https://doi.org/10.1145/2642937.2648624

[11] Sandra Schröder and Matthias Riebisch. 2017. Architecture Conformance Checking with Description Logics. In *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings (ECSA '17).* ACM, New York, NY, USA, 166–172. https://doi.org/10.1145/3129790.3129812

[12] Sandra Schröder, Matthias Riebisch, and Mohamed Soliman. 2016. *Architecture Enforcement Concerns and Activities - An Expert Study.* Springer International Publishing, Cham, 247–262. https://doi.org/10.1007/978-3-319-48992-6_19

[13] Rolf Schwitter. 2010. Controlled Natural Languages for Knowledge Representation. In *Proceedings of the 23rd International Conference on Computational Linguistics: Posters (COLING '10).* Association for Computational Linguistics, Stroudsburg, PA, USA, 1113–1121. http://dl.acm.org/citation.cfm?id=1944566.1944694

[14] Rolf Schwitter, Kaarel Kaljur, Anne Cregan, Catherine Dolbear, and Glen Hart. 2008. A Comparison of three Controlled Natural Languages for OWL 1.1. In *In 4th OWL Experiences and Directions Workshop (OWLED 2008 DC.*

[15] Evren Sirin. 2010. Data Validation with OWL Integrity Constraints. In *Web Reasoning and Rule Systems*, Pascal Hitzler and Thomas Lukasiewicz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 18–22.

[16] Ricardo Terra and Marco Tulio Valente. 2009. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience* 39, 12 (2009), 1073–1094. https://doi.org/10.1002/spe.931