Architecture Conformance Checking with Description Logics

Sandra Schröder and Matthias Riebisch
Universität Hamburg, Department Informatics
Vogt-Kölln-Straße 30
Hamburg, Germany
{schroeder,riebisch}@informatik.uni-hamburg.de

ABSTRACT

Today, a lot of commercial and open source tools exist allowing to describe the intended architecture and to check architecture conformance automatically in order to detect and eventually minimize erosion. Unfortunately, those tools are restricted in terms of which architecture concepts can be used in order to describe the intended architecture. Additionally, those approaches lack an appropriate formal foundation in terms of a well-defined syntax and semantic. Nevertheless, this is a crucial requirement for automatic support in architecture conformance checking. In this paper, we propose a formal approach enabling a) the definition of an architecture concept language that allows to capture the most important architecture concepts and their related architecture rules, b) the validation of the consistency of this language, c) the mapping of architecture concepts to source code and capturing this mapping in an explicit language in order to d) check architecture conformance. We also present an evaluation of the approach using the Common Component Modeling Example (CoCoME) case study in order to demonstrate the applicability.

CCS CONCEPTS

• **Software and its engineering → Software architectures**; *Software evolution*;

KEYWORDS

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

ACM Reference format:
Sandra Schröder and Matthias Riebisch. 2017. Architecture Conformance Checking with Description Logics. In *Proceedings of ECSA '17, Canterbury, United Kingdom, September 11–15, 2017,* 7 pages.
DOI: 10.1145/3129790.3129812

1 INTRODUCTION

Architecture conformance checking [11] has been recognized as an important mean to detect and control software architecture erosion [14]. Due to the inherent complexity of software systems, the manual validation of architecture rules is not feasible at least for larger systems. Consequently, this process must be supported by automatic methods. A lot of approaches and tools have been developed in order to support this process such as Sonargraph [4], HUSACCT [15], or DCL [19]. Nevertheless, we only found one approach that uses a formalism, namely first-order logic, to formalize and validate architecture rules [8, 10]. We think it is a promising and necessary approach and we wonder what could be the reasons why it was not pursued by the software architecture community. On the one hand, a major reason could be that the formalism might be too complex for architecture rule definition and is therefore not really usable in practice. On the other hand, its undecidability might make the checking of those rules once defined inefficient in terms of computational costs. This is also a crucial criterion for applicability in practice.

Other available approaches might provide a more usable architecture rule definition. However, they are mostly not expressive enough for flexible and rich architecture rule definition and checking. However, we think that such an approach based on well-defined syntax and semantics is necessary to support architecture conformance checking efficiently. We therefore aim to develop a practically usable conformance checking approach providing flexible and richer architecture rule definition and checking. As a first step, we provide in this paper a formal foundation of this approach.

We recognized that description logics [5] provide a suitable formalism for architecture rule definition and their validation that may provide additional advantages and may overcome the limitations of other approaches. We conducted a feasibility analysis of this approach and present preliminary results of our study at hand.

The paper is structured as follows: In Section 2 we provide a formal foundation about description logic and present an overview of our approach. Next, in Section 3, we present a feasibility study of this approach based on selected architecture rules from the Common Component Modeling Example (CoCoME). Simultaneously, we provide challenges for further development and evaluation of the approach. Section 5 concludes the paper.

2 APPROACH

2.1 Background on Description Logics

Before presenting our approach, we introduce the basics and core elements of the most common description logic language $\mathcal{SROIQ}$. It constitutes the most expressive description logic with the advantage that reasoning is decidable. Description logics are a family of the logic based knowledge presentation formalisms. They have a formally defined syntax and semantics and therefore allow for a precise specification of concepts and their properties in a domain of interests. In description logics, the terminology of the application domain is formalized using *concepts*, *roles*, and *individuals*. In the following, N_C, N_R, and N_I denote the disjoint set of concept

ECSA '17, Canterbury, United Kingdom
 978-1-4503-5217-8/17/09…$15.00
DOI: 10.1145/3129790.3129812

names, role names, and individual names respectively. The triple (N_C, N_R, N_I) is called the signature (or also the vocabulary):

- 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 predicates.
- Roles are relations between concepts. They can be thought of as binary predicates.
- The set N_I contains all names used to denote singular entities. They represent constants in the formalism.

Using the $\mathcal{SROIQ}$ formalism, knowledge bases can be created using three building blocks, namely the so-called TBox and ABox. In the following, we will describe the main characteristics of the TBox and ABox blocks briefly.

$\mathcal{SROIQ}$ allows to specify concept descriptions such as basic concept descriptions, universal restriction, existential quantification, and qualified number restrictions. Concept expressions are inductively defined as follows:

- Basic concepts descriptions can be formed according to the following syntactical rules: $\top$ and $\bot$ are concept expressions, 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.

$\mathcal{SROIQ}$ allows for the definition of general concept inclusion (GCI) axioms (also called subsumption axioms). Those have the form $C \sqsubseteq D$ where C and D are concepts. This axiom type can be thought as a *is-a* relationship implying a hierarchical relationship between concepts. A finite set of GCIs is the so-called terminological box (TBox) of the $\mathcal{SROIQ}$ knowledge base. The assertional box of the knowledge base (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 expression and r is a role.

Description logics provide decidable reasoning services. Highly optimized reasoning algorithms have been developed and have shown that tableaux algorithms, even for highly expressive description logics, lead to a good performance even on large knowledge bases [5]. Our approach could benefit greatly from those efficient reasoning algorithms in terms of computational costs since we apply them for the validation of architecture rules. This allows for architecture conformance checking even in complex software systems.

Description logics are well-supported by tools and standards, as they are the formal foundation for ontology languages and the semantic web, respectively. For the semantic web a lot of mature standards such as the Web Ontology Language (OWL) [2] or the Semantic Web Rule Language (SWRL) [1] have been developed in order to support the definition of ontologies. That is why, there is appropriate tool support for ontology creation and reasoning available that can be used out-of-the-box for implementing our proposed solution.

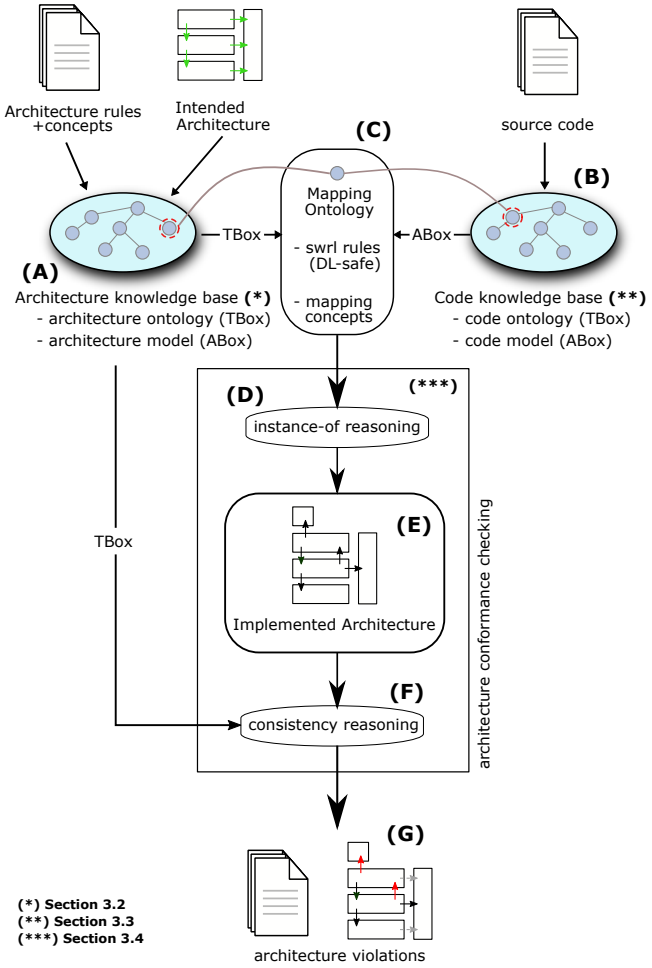


Figure 1: Overview of the architecture conformance checking process based on description logics.

Ontologies based on the description logic formalism can be defined in modules. Each module is related to a specific type of software system containing a set of architecture concepts and related architecture rules. For example, we can define an ontology for component-based software architectures and another ontology for event-based software architectures. A software system can then be checked separately against each module. In this way, it is possible to validate which types of rules are violated from which type of system, e.g. we can make statements like: the system violates a lot of rules defined for event-based architectures, but it conforms to the rules of component-based software architectures. Another advantage of this modularization is that an ontology can import concepts from another ontology. This allows for reuse and refinement of concepts from an existing ontology. The software architect can define a new concept, that inherits the properties and rules of the existing concepts, but maybe has additional rules.

2.2 Overview of the Approach

Using description logic, we are able to define a so-called *ontology* for the description and development of software architectures in a formal and machine processable way. The ontology is the basis for an *architecture concept language*, with which the software architect describes the software architecture model in terms of so-called *architecture concepts*. In Figure 1 an overview of the approach and its main building blocks is given. In this process, three important artifacts need to be created by the architect:

(1) the *architecture concept language* capturing the most important architecture concepts and rules by which
(2) the *concrete software architecture model* can be described and
(3) an *architecture-code-mapping* describing how architecture concepts correspond to source code elements.

The architecture concept language - i.e. the architecture ontology - describes a platform independent formalization of the most important concepts and their corresponding rules needed to describe the software architecture. Architecture concepts directly map to atomic concepts (N_C). Relationships between architecture concepts are modeled as roles (N_R). Concepts and roles are further formalized using complex concept expressions (see Section 2.1) in order to express architecture rules. Those descriptions form the TBox of the architecture ontology. The concrete individuals, i.e. the ABox, of the concepts form the concrete architecture model. Together, the ontology and the software architecture model constitute the architecture knowledge base **(A)**. Another important artifact is the code knowledge base **(B)** encompassing the ontology representing concepts of the code (for example of the Java programming language) and individuals of the code ontology representing the actual source code, i.e. the code model. The mapping ontology constitutes the core element of the architecture conformance checking process **(C)**. It contains logical statements describing how the architecture concepts and the code concepts from the respective ontologies are connected with each other. For this, it unites the axioms of architecture concepts located in the TBox of the architecture ontology with the TBox of the code ontology and the assertions corresponding to the model of the actual source code (ABox) for which the conformance will be checked. The mapping ontology defines separate mapping concepts and mapping rules. The mapping rules in our approach are described as Horn-like logical rules using the syntax of the Semantic Web Rule Language (SWRL rules) [1]. As we stated in Section 2.1, we need to apply so-called DL-safe rules in order to preserve decidability. The reasoner Pellet [18] applied in our approach supports reasoning with DL-safe rules. In Section 3 we show the general syntax of those rules in more detail and describe concrete examples of mapping rules for the case study. Rules are applied by invoking reasoning services. The instance-of relation is calculated for the source code elements of the code model **(D)**. This has the effect that code elements are represented in terms of architecture concepts, i.e. those concepts that are defined in the TBox of the architecture ontology. This means, the implemented architecture is extracted from the code **(E)**. The implemented architecture can then be checked against the axioms defined in the TBox of the architecture ontology using the consistency reasoning service **(F)**. This is the actual architecture conformance checking process. An inconsistency with respect to a TBox axiom in the architecture ontology is considered an architecture violation **(G)**.

3 EVALUATION

In this section, we demonstrate our approach using the Common Component Modeling example (CoCoME) [9]. We have implemented our approach using the Protégé tool [12], an editor for creating and storing ontologies based on the Web Ontology Language (OWL) [2] standard. $\mathcal{SROIQ}$ provides the formal basis of this language (see Section 2.1). The tool additionally integrates different reasoners, such as Pellet [18] that we have used for the reasoning tasks.

3.1 The Architecture Rules of CoCoME

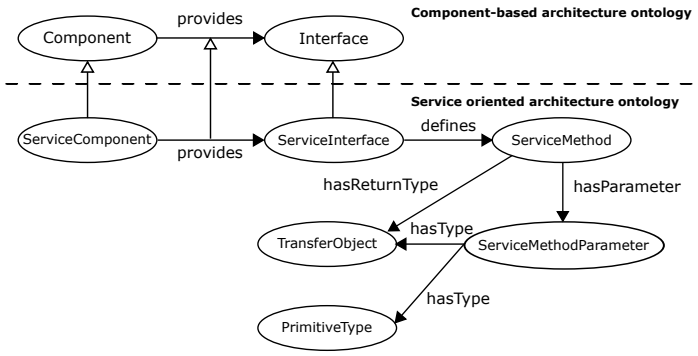
We have applied the architecture rules as defined in [8].

- **Rule 1:** *"The access from the GUI layer to the application layer is limited to calls of methods defined in service components; these are components providing at least one service interface."*
- **Rule 2:** *"A service interface is an interface defining service methods only. These are methods using only transfer object classes and primitive types for parameters and return value types. Returned objects are always created during the call of the service method."*
- **Rule 3:** *"Transfer objects refer only to other transfer objects or data entities of primitive types."*

In order to define the ontologies needed for the architecture conformance checking process, we applied the process of ontology design presented in [13]. There, five necessary steps are depicted. For example, a process is described how to extract the relevant concepts for the ontology which is a crucial step for designing our architecture concept language. We do not present this process in detail here, but refer to [13].
In the following we present an excerpt of the formalization.

3.2 Formalizing Architecture Concepts of CoCoME

In Figure 2 we illustrate how the architecture rules described in the previous section can be formalized. The left part of Figure 2 visualizes two ontologies. The ontology describing the concept of service-oriented architectures is presented using the Manchester OWL syntax [3] as it is used in the Protégé environment to describe ontologies (see right part of Figure 2). In this syntax, mathematical symbols such as $\forall$, $\exists$ or $\neg$ (for universal, existential restriction, and negation, respectively) have been replaced by more intuitive keywords like `only`, `some`, and `not`.
On the left part of Figure 2, the graph on the top shows a simple and general ontology for component-based architectures. The main concepts in those type of systems are components and interfaces. That is why we added them to the TBox of this ontology, namely as *Component* and *Interface*. The two roles *provides* and *requires* formalize the binary relation between components and interfaces: a component provides or requires an interface. Second, this ontology is further extended by the ontology describing concepts for service-oriented architectures necessary for formalizing the architecture




```
Class: ServiceComponent
 SubClassOf: Component and provides min 1 ServiceInterface
 DisjointWith: ServiceInterface, ServiceMethod, ...

Class: ServiceInterface
 SubClassOf: Interface and defines only ServiceMethod
 DisjointWith: ServiceComponent, ServiceMethod, ...

Class: ServiceMethod
 SubClassOf: hasParameter only (hasType only (PrimitiveType
                or TransferObject))
 SubClassOf: hasReturnType only PrimitiveType
                or TransferObject)
 DisjointWith: ServiceComponent, ServiceInterface, ...
```

Figure 2: An excerpt from the description logic based formalization of the CoCoME case study. Left: Visualization of the component-based architecture ontology and the service-oriented architecture. Right: Formalization of the service-oriented ontology using the OWL Manchester syntax [3].

rules of CoCoME. In total, six concepts and five roles are defined in this ontology. The necessary concepts are directly extracted from the architecture rule description presented in the previous section. Similarly, the roles between concepts are extracted and/or refined, namely *provides*, *defines*, *hasReturnType*, *hasType*, *hasParameter*. The keyword `Class` introduces a concept definition for the concept *ServiceComponent*. This is also called a class axiom or class definition in OWL. A class axiom corresponds to an architecture rule. The concept *ServiceComponent* subclasses (general concept inclusion, keyword `SubClassOf`) a complex concept expression. The expression `Component` **`and`** `provides` **`min`** `1 ServiceInterface` is an intersection (keyword and) of the concept *Component* and another complex concept description limiting the *provides* role. It describes a set of instances, that are namely a component and provide at least one interface. This means, that it reuses the concept *Component* from the ontology for component-based software architectures and defines additional rules. In the service-oriented architecture ontology, those interface must be an instance of the *ServiceInterface* concept. We use the at-least restriction ($\geq nR.C$, keyword `min <number>`) to limit the *provides* role to this type of interface, so that there must be at least one relationship between an individual of a service component and a service interface. The *ServiceInterface* concept is formalized analogously. In order to restrict the *define* role on individuals of the concept *ServiceMethod* (corresponding to the rule: *"A service interface is an interface defining service methods only"*), we make use of the universal restriction on roles ($\forall$, keyword `only`). For the concept *ServiceMethod* two class axioms are defined that relate to the corresponding architecture rules. Three roles are important here and need to be limited in order to implement the rules, namely *hasParameter*, *hasType*, and *hasReturnType*. In the first axiom, we formalize the rule that service methods are only allowed to have *ServiceMethodParameter* as a parameter[1]. The second axiom ensures that only transfer objects and primitive types are allowed as return types. It is important to note that all those concepts need to be defined as disjoint concepts. Otherwise, we get undesirable side effects during reasoning, e.g. an individual of the concept *ServiceInterface* is inferred to be also a member of the concept *ServiceComponent*.

3.3 Formalizing the Code and the Architecture-Code-Mapping

In order to formalize the code as an ontology, we use the FAMIX meta model [6] as reference in order to derive essential concepts and roles for the ontology. FAMIX provides a common source code meta model in order to represent facts about a 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. We refer to [6] for a deeper description of this meta model. The classes of the FAMIX meta model can be mapped directly to concepts of the code ontology as we will demonstrate in the following. Figure 3 a) shows a code snippet from a service-oriented interface of CoCoME, Figure 3 b) an excerpt from the FAMIX meta model, and Figure 3 c) the corresponding individuals (ABox) describing the code using concepts from the FAMIX-based code ontology. In order to express this snippet, we need the elements *Class* (note that we have named the concept *FamixClass* in our ontology in order to avoid conflicts with the internal *Class* concept of OWL), *Method*, *Inheritance*, *Namespace*, *PrimitiveType*, and *Parameter* from the FAMIX meta model. In order to express that a method is contained in a class, the role *parentType* is defined. In Figure 3 c) we express that the method `getStoreStockReport` is contained in the interface `IReporting` using the role *parentType*. According to the FAMIX model, we defined inheritance between two classes explicitly as a concept and not as a role. In order to express that a class extends another class or interface, additional roles are defined, namely *hasSuperClass* and *hasSubClass* as it is shown in Figure 3 c). The roles *famixHasName*, *famixHasModifier*, and *isInterface* are so-called data properties that link individuals to data values. The first two roles link individuals to strings, whereas the latter links the individual to a boolean value. We use *famixHasName* in order to assign a name to a code element. Since interfaces are not modeled as an explicit element, but as a property of the *Class* element in the meta model (see Figure 3 b)), we use the *isInterface* role to denote a class as an interface.

[1]To shorten the listing in Figure 2 for visualization, we use the compact definition `hasType` **`only`** `(PrimitiveType` **`or`** `TransferObject)` and spare a separate definition of the *ServiceMethodParameter* concept

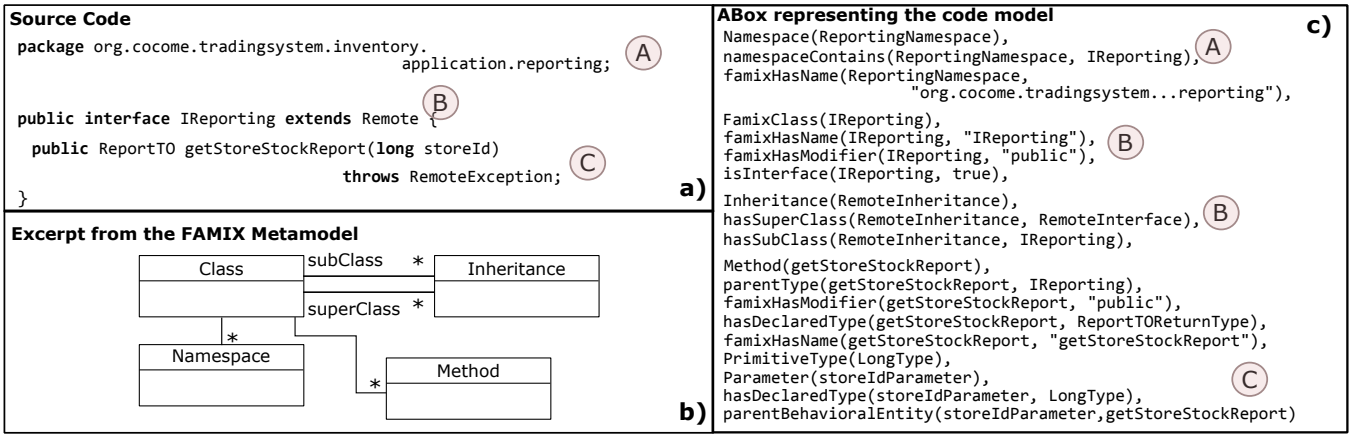


Figure 3: Representing the code model as individuals of the code ontology based on the FAMIX metamodel. Left: A code snippet from CoCoME. It implements the provided interface of the service component "Reporting" in the service-oriented layer. Right: The individuals representing the code snippet.

In the next step, we need to express the architecture-code-mapping. The mapping is also described as a separate ontology which imports the architecture and the code ontology. For each architecture concept, a mapping concept needs to be specified. For that, we define a separate concept and a corresponding rule using the syntax of the Semantic Web Rule Language (SWRL) [1] that derives the mapping concept based on existing knowledge about the architecture and the code. Rules consist of an implication between an antecedent (body) and a consequent (head). 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"*, written as $b_1, b_2, \ldots, b_n \rightarrow a$. The body ($b_1, b_2, \ldots, b_n$) and the head ($a$) consist of atoms. The body is a conjunction of atoms. Atoms can be of the form `C(x)` or `P(x,y)`, where `C` is a concept (or class in OWL language) and `P` is a role (or property in OWL language). `x` and `y` are variables, individuals or data values. During reasoning, values are bound to those variables and it is tested if an atom holds for this value. An atom `C(x)` holds if `x` is an instance of the concept `C` or `P(x,y)` holds if `x` is related to `y` by role `P`. Intuitively, in our approach, a stands for the mapping concept that we want to infer for a specific individual, whereas $b_1, b_2, \ldots, b_n$ are atoms corresponding to individuals from the architecture and the code model that must hold in order to infer the mapping concept. We exemplary show rules for the mapping of service components and their provided interfaces to the code. In CoCoME each component is mapped to a Java package where its identifier contains the name of the corresponding component. For this, we define the concept *MappedComponentByPackageName*. Using general concept inclusion and concept equivalence, we can connect the concepts *MappedComponentByPackageName*, *ServiceComponent* and *Namespace* as follows:

$$MappedComponentByPackageName \equiv ServiceComponent \quad (1)$$

$$MappedComponentByPackageName \sqsubseteq Namespace \quad (2)$$

The corresponding SWRL rule for the mapping is formalized as shown in Listing 1.

Listing 1: A mapping rule in SWRL syntax for mapping service components to Java packages.

```
Namespace(?namespace),
famixHasName(?namespace, ?nameOfNamespace),
ServiceComponent(?component),
hasName(?component, ?nameOfComponent),
swrl:contains(?nameOfNamespace,?nameOfComponent)
-> MappedComponentByPackageName(?namespace),
hasName(?namespace,?nameOfComponent)
```

Identifiers prefixed with ? are variables. This rule states the following: There must be an individual `?namespace` from the *Namespace* concept, i.e. the atom `Namespace(?namespace)` must hold). This individual must have the specific name `?nameOfNamespace`. There must be an individual `?component` named `?nameOfComponent`. This name must be contained in the identifier of the namespace. For this, SWRL provides built-ins. In this case, we use the built-in function for string comparison, namely `swrl:contains`. If individuals can be found satisfying the rule body, then we can infer that the individual `?namespace` is also an individual of the class *MappedComponentByPackage*. Since we stated that *MappedComponentByPackageName* and *ServiceComponent* are equivalent, the reasoner infers that this individual is also an instance of the concept *ServiceComponent* additionally with the relation *hasName*, a role from the architecture ontology relating a component with a string value constituting the component's name. In this way we can also infer which specific individual of type *ServiceComponent* the Java package refers to. The mapping of the provided interface of the inferred component instance is similarly formalized. For this, we define another concept *MappedInterface* with the corresponding rules in order to infer the instances of this concept. The provided interface is mapped to the public Java interface contained in the package that we mapped before to the corresponding component. Additionally, we need to lift the code-level relationships to architecture-level relationships. For example, we map the contain relationship between packages and classes/interfaces to the provide relationship from the architecture ontology as shown in Listing 2.

Listing 2: A mapping rule in SWRL syntax for mapping the contain relationship to the provides relationship.

```
MappedComponentByPackage(?mappedComponent),
FamixClass(?interface),isInterface(?interface, true),
famixHasName(?interface, ?name),
ServiceInterface(?serviceInterface),
has_name(?serviceInterface, ?interfaceName),
famixHasModifier(?interface, "public"),
namespaceContains(?mappedComponent, ?javaInterface)
−> provides(?mappedComponent, ?javaInterface)
```

3.4 Performing Architecture Conformance Checking

The implemented architecture can now be checked for architecture conformance using reasoning algorithms. This means that the architecture concepts extracted from the code are checked for concept consistency. We use the reasoning process in order to check if individuals fulfill the concept axioms stated in the architecture ontology. In order to demonstrate this, we artificially added an architecture violation to the code snippet presented in Figure 3. We therefore changed the signature of the service method in the service interface `IReporting`, so that it returns the code-level type `AComplexType` that is a Java class. Thus, `AComplexType` is an individual belonging to the *FamixClass* concept. After invoking the reasoner, this results in an inconsistency. Reasoners allow to list explanations helping to find reasons for inconsistencies. In this way, the architect not only has the information about *which* violations occurred, but also *why* architecture rules are violated. We think that this is a major advantage in contrast to existing approaches. In those explanations, the concept axioms which are violated by individuals are highlighted. An excerpt of such an explanation is shown in Listing 3. Blue keywords depict OWL specific keywords, whereas keywords highlighted in red are violated axioms. The result can be explained as follows: Methods in the code model are mapped to the *ServiceMethod* concept, if they are contained in a public java interface that was mapped to the *ServiceInterface* concept before. For those methods, the code-level role *hasDeclaredType* is mapped to the architecture-level role *hasReturnType* (line 5). The declared type of the code-level method `getStoreStockReport` is `AComplexType` (line 2) which is mapped to the architecture-level concept *ComplexType* (line 3). We denote this individual as *complexType1*. However, this violates the axiom in line 4. The only way to resolve this inconsistency, the reasoner could infer that *complexType1* is a subtype of *TransferObject* or *PrimitiveType* or is equivalent with them. But this is not possible, since we stated that *ComplexType*, *TransferObject*, and *PrimitiveType* are disjoint with each other (line 7), an inconsistency is detected by the reasoner. This corresponds to an architecture violation. This violation has also consequences for other axioms. Since the method does not satisfy the rules of a service method and is then, by definition, not a service method anymore, the rules regarding service interfaces (line 9-10) are violated. A service interface is required to only define service method, but there is a method that does not satisfy this rule. Consequently the service interface cannot satisfy its rules.

Listing 3: Output of the reasoning process

```
1  AComplexType famixHasName "AComplexType"
2  getStoreStockReport hasDeclaredType AComplexType
3  FamixClass(?c), famixHasName(?c,"AComplexType") −> ComplexType(?c)
4  ServiceMethod SubClassOf hasReturnType only (PrimitiveType or TransferObject)
5  MappedServiceMethod(?m), hasDeclaredType(?m,?rt) −> hasReturnType(?m,?rt)
6  AComplexType Type FamixClass
7  DisjointClasses: ComplexType, PrimitiveType, ServiceComponent, TransferObject...
8  Reporting provides ReportingIf
9  ReportingIf defines getStoreStockReport
10 ServiceInterface SubClassOf defines only ServiceMethod
```

3.5 Challenges and Further Evaluation of the Approach

During the formalization and validation of the rules, we encounter some challenges, especially regarding **Rule 1**. If a service component is missing to provide a service interface, the reasoner would not infer an inconsistency. This is because description logics are based on the so-called open-world assumption [5]. The open-world assumption intuitively states that anything not explicitly expressed is unknown. This means, that ontologies use a form of underspecification as a means of abstraction. A reasoner will then simply add an individual to the existing assertions and proceed with making inferences. We solve this conflict by limiting the universe to the known individuals from the ABox of the architecture and code knowledge base, respectively. This hinders the reasoner to add anonymous individuals during the inferencing process. It needs to be further investigated how suitable this work-around is especially with regards to software systems containing numerous components and interfaces. Another important point is the comprehensibility of the explanations given by the reasoners (see Listing 3). In order to make them valuable for software architects, postprocessing of the explanations in case of inconsistencies is necessary. We need to find an appropriate way to present the explanation to the software architect in a more comprehensible way, so that architecture violations can be easily extracted from them.

As stated in the introduction, an architecture concept language should be usable to some extent in order to be applicable in practice. In the next steps, we plan to investigate, how the approach can be adapted in order to make it more usable. For this, we consider controlled natural languages integrating well with description logics [17]. In this way, rules can be expressed in a more natural way. However, there might be a trade-off between usability of the specification language and its expressiveness concerning the variety of concepts and rules that can be defined. This trade-off additionally needs to be investigated in the future. This should especially be validated in industry. There, it should be tested how easily developers and architects can apply the approach in order to define the necessary rules and whether all important architecture concepts and rules can be expressed with the formalism.

Another interesting aspect that needs to be considered is the performance of the approach. In order to enable its applicability in practice, the approach needs to be able to check rules even on large code bases. We plan to investigate if decidable reasoning services give a benefit in terms of performance as stated in Section 2.

4 RELATED WORK

Several tools and approaches have been developed enabling architecture conformance checking such as [4, 7, 15, 19]. However, the existing tools restrict the architect in the way how the intended architecture can be specified and which types of architecture rules

can be defined and validated. This means that the architect has to use an architecture specification language which is fixed in terms of the architecture concepts it provides and is not able to extend this language with new or refined concepts that better fit the current requirements. That is why, the architect is forced to translate his concepts to the provided concepts of the language. We discuss this using the tool Sonargraph, since it is mostly used in industry for formalizing architecture rules as we found out in our recent study [16]. The tool provides concepts like layers, layer groups, vertical slices, vertical slices groups and subsystems in order to describe the intended architecture. In the case of **Rule 1** the layer concept provided by Sonargraph is sufficient in order to model the layer constraints (*"The access from the GUI layer to the application layer is limited to calls of methods defined in service components"*). However, we cannot directly define specific architecture concepts, such as service components and service methods with their respective rules (as it is necessary for the second part of **Rule 1**, for **Rule 2** and **Rule 3**). For example, the rule that service methods are only allowed to return primitive types or transfer object cannot be easily expressed with Sonargraph. This can at most be formalized by defining dependency rules between methods and the specific types. Consequently, the original concepts defined by the architect and their respective rules are not directly visible in the tool and some of them cannot be mapped at all. That is why, the design intent of the software architecture gets lost and which type of architecture concept and its respective rule is violated is hardly comprehensible. Our approach allows to directly express the intended architecture in terms of the necessary and most important concepts and their corresponding rules. Moreover, in order to extend the language with new concepts, the whole tool infrastructure would have to be changed in order to support new concepts in existing approaches. In our approach, this change is much more lightweight due to the use of ontologies.

5 CONCLUSION AND FUTURE WORK

In this paper, we presented an approach that allows to flexibly define a concept language for describing the intended software architecture. The approach has the advantage of explicitly capturing the concepts a software architecture consists of and is not restricted to concepts given by a fixed architecture description language. Additionally, it allows for the definition of concept related rules, i.e. architecture rules, against which the source code can be checked in order to detect architecture erosion. We evaluated the feasibility of the approach using the CoCoME case study.

Some aspects of our approach have to be investigated further. For example, we only formalized a small part of CoCoME and further need to evaluate if the expressiveness of description logics really suffices for formalizing architecture rules. For this, we plan to formalize architecture rules from other software systems. We also need to think about how to handle the open-world behavior of description logics. In some situations, as explained in Section 3, the reasoner does not find an inconsistency, when information is missing in the architecture and code models.

However, we have shown that it is possible to formalize and validate architecture rules using description logics. We claim that the concepts driven by architecture decisions and their mapping to the source code should explicitly be captured and clarified as an integral part of the software architecture design process. This supports the clear communication of the architectural concepts between the architect and development team and additionally the preservation of the knowledge about those concepts and the corresponding rules over time in order to support software evolution. That is why, further elaboration regarding the realization of the approach and addressing the remaining challenges is promising. We conclude that description logics provide a suitable formalism and that existing and future approaches could benefit from integrating this work.

REFERENCES

[1] 2004. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. (May 2004). https://www.w3.org/Submission/SWRL/

[2] 2012. OWL 2 Web Ontology Language Document Overview (Second Edition). (December 2012). https://www.w3.org/TR/owl2-overview/

[3] 2012. OWL 2 Web Ontology Language Manchester Syntax (Second Edition). (December 2012). https://www.w3.org/TR/owl2-manchester-syntax/

[4] 2017. Sonargraph-Architect. (2017). https://www.hello2morrow.com/products/sonargraph/architect9

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

[6] Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. 2011. *MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family*. Technical Report.

[7] S. Duszynski, J. Knodel, and M. Lindvall. 2009. SAVE: Software Architecture Visualization and Evaluation. In *2009 13th European Conference on Software Maintenance and Reengineering*. 323–324. DOI: http://dx.doi.org/10.1109/CSMR.2009.52

[8] Sebastian Herold. 2011. *Architectural Compliance in Component-Based Systems*. Ph.D. Dissertation. Clausthal University of Technology. http://www.dr.hut-verlag.de/978-3-8439-0109-3.html

[9] Sebastian Herold, Holger Klus, Yannick Welsch, Constanze Deiters, Andreas Rausch, Ralf Reussner, Klaus Krogmann, Heiko Koziolek, Raffaela Mirandola, Benjamin Hummel, Michael Meisinger, and Christian Pfaller. 2008. The Common Component Modeling Example. Springer-Verlag, Berlin, Heidelberg, Chapter CoCoME - The Common Component Modeling Example, 16–53. DOI: http://dx.doi.org/10.1007/978-3-540-85289-6_3

[10] 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. DOI: http://dx.doi.org/10.1109/EDOC.2013.17

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

[12] Mark A Musen. 2015. The Protégé project: A look back and a look forward. *AI matters* 1, 4 (June 2015), 4–12.

[13] Natalya F Noy, Deborah L McGuinness, and others. 2001. Ontology development 101: A guide to creating your first ontology. (2001).

[14] 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.

[15] 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. DOI: http://dx.doi.org/10.1145/2642937.2648624

[16] Sandra Schröder, Matthias Riebisch, and Mohamed Soliman. 2016. Architecture enforcement concerns and activities-an expert study. In *Proceedings of the 10th European Conference on Software Architecture (ECSA2016)*. Springer, 247–262.

[17] Rolf Schwitter and Marc Tilbrook. 2006. Let's talk in description logic via controlled natural language. In *Proceedings of the 3rd Int. Workshop on Logic and Engineering of Natural Language Semantics (LENLS 2006)*.

[18] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. 2007. Pellet: A Practical OWL-DL Reasoner. *Web Semant.* 5, 2 (June 2007), 51–53. DOI: http://dx.doi.org/10.1016/j.websem.2007.03.004

[19] 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. DOI: http://dx.doi.org/10.1002/spe.931