Semi-automatic Architectural Pattern Identification and Documentation Using Architectural Primitives

Thomas Haitzer*, Uwe Zdun

Software Architecture Research Group, University of Vienna, Vienna, Austria

Abstract

In this article, we propose an interactive approach for the semi-automatic identification and documentation of architectural patterns based on a domain-specific language. To address the rich concepts and variations of patterns, we firstly propose to support pattern description through architectural primitives. These are primitive abstractions at the architectural level that can be found in realizations of multiple patterns, and they can be leveraged by software architects for pattern annotation during software architecture documentation or reconstruction. Secondly, using these annotations, our approach automatically suggests possible pattern instances based on a reusable catalog of patterns and their variants. Once a pattern instance has been documented, the annotated component models and the source code get automatically checked for consistency and traceability links are automatically generated. To study the practical applicability and performance of our approach, we have conducted three case studies for existing, non-trivial open source systems.

Keywords: Software Architecture, Architectural Component Views, Design and Architecture Understanding, Architectural Pattern

1. Introduction

During maintenance and evolution of a software system, a deep understanding of the system's architecture is essential. This knowledge about a system's architecture tends to erode over time [1] or even get lost. In a recent study Rost et al. [2] found that architecture documentation is frequently outdated, updated only with strong delays, and inconsistent in detail and form. They also found that developers prefer interactive (navigable) documentation compared to static documents. This also reflects our personal experiences as well as those of others. For instance, our colleague Neil Harrison shared the following story from his experiences with large-scale industrial systems

*Corresponding author

Email addresses: thomas.haitzer@univie.ac.at (Thomas Haitzer), uwe.zdun@univie.ac.at (Uwe Zdun)

URL: http://informatik.univie.ac.at/thomas.haitzer (Thomas Haitzer), +43-1-4277-78521 (Thomas Haitzer), +43-1-4277-8-78521 (Thomas Haitzer), http://informatik.univie.ac.at/uwe.zdun (Uwe Zdun)

(shortened): “Once upon a time I worked on a large system that was already a few years old. It had a well-defined architecture. When I started, I was given copies of three or four documents that described the architecture. In addition, I watched several videotapes in which the architects described the architecture. As a result, I gained a good understanding of the architecture of the system. After a few years, I left the project to work on other things. But several years later I returned. The system was still being used and was under active development. Of course, it had changed greatly to add new capabilities and support changes in technology. Underneath it all, the original architecture was largely intact, but it was much more obscure. I wanted to refresh my architectural memory, so I asked around for the original memos and videotapes. Nobody had even heard of them. Critical architectural knowledge had been lost. People were actually afraid to change the original code, because they didn’t understand how it worked.”

Software architecture documentation or, in case of lost architectural knowledge, software architecture reconstruction [3] techniques can be used to (re)establish the proper architectural documentation of the software system. An essential part of today’s architectural knowledge is information about the patterns used in a system’s architecture. Patterns can be seen as building blocks for the composition of a system’s architecture [4, 5]. This is especially valid for architectural patterns or styles which describe a system’s fundamental structure and behavior [6]. A considerable number of software architecture reconstruction approaches support software pattern identification [5, 7, 8]. Most of these approaches (see e.g. [9, 10, 7, 11]) focus on automatically detecting design patterns in the source code. Such pattern identification approaches are often restricted to design patterns that were identified by Gamma et al. [12] (GoF patterns). Architectural patterns, in contrast, convey broader information about a system’s architecture as they usually are described at a larger scale than GoF patterns.

There are a number of important problems in automatic pattern identification in general and especially in architectural pattern identification. Existing approaches often only focus on the task of identifying a system’s design patterns while the documentation of the reconstructed patterns and the future evolution of the system are not considered (which is just as essential as identifying an architectural pattern).

In addition, architectural patterns are often much harder to detect directly in the source code than GoF design patterns as there is often a large number of classes involved in the implementation of the pattern and the variations between different instances of the patterns are very large. As a consequence of the large number of involved classes there is a possibly huge search space for these patterns that grows with every class and increases execution times [3].

A big problem of pattern identification is the variability in pattern implementations. Only a very few pattern identification approaches consider pattern variations at all, and they are usually focused on GoF design patterns only [13, 14]. For instance, hardly any implementation of a system strictly adheres to the Layers pattern [4] as described in the textbook, but a huge number of systems are designed based on Layers. To give a concrete example, in the definition of the Layers pattern, a layer only has access to the functionality provided by the layer below it. However, this rule is often violated for cross-cutting concerns like performance, security, or logging. As a consequence, many layered architectures contain parts that do not strictly adhere to the Layers pattern. In

addition to this, the Layers pattern suggests but does not in any way enforce clean interfaces between the layers. For these reasons, it is hard to automatically detect architectural patterns like Layers.

Another problem of automatic pattern identification is the accuracy of the approaches, which is often not sufficient. That is, some approaches treat pattern instances they find as candidates [14]. However the likelihood of false positives increases with system size and can lead to precision values around 40 percent [10] which means that 60 percent of the found pattern instances are false positives. This requires substantial manual effort to review the found pattern instances.

In the light of the aforementioned problems, we formulated the following research questions:

RQ1 How far can a semi-automatic architectural pattern approach go toward the goal of identifying the patterns in architectural reconstruction?

RQ2 How far can a semi-automatic architectural pattern approach go toward the goal of maintaining documented patterns during the further evolution of a reconstructed architecture?

RQ3 In how far are the concepts and tools applicable in existing real-life systems?

RQ4 How efficient are the actual pattern instance matching algorithms that are based on primitives?

RQ5 Are primitives and an adaptable pattern catalog adequate means to handle the variability inherent to architectural patterns?

The main contributions of this article are, first, to suggest a novel semi-automatic architectural pattern identification approach that tackles the aforementioned problems that arise during the documentation and evolution of architectural patterns like the variability inherent to patterns, consistency between the documented architecture and the source code, and the large number of source code artifacts that are related to the implementation of architectural patterns. Second, we show the approach's feasibility in terms of tool support (in the context of three open source case studies), and to study the performance of the approach (also in the context of these cases). We aim to assist the software architect during the reconstruction of architectural knowledge as well as supporting the architect in the documentation of the reconstructed architectural knowledge. After the architectural knowledge has been reconstructed and documented with our approach, we support the software architect in keeping the created architectural documentation in sync with the source code of the application. As Clements et al. [15] state, a strong architecture is only useful if it is properly documented in order to allow others to quickly find information about it.

Our proposed solution is an interactive approach for the semi-automatic identification and documentation of architectural patterns based on a set of Domain Specific Languages (DSLs). It consists of the following main components:

- *Architecture Abstraction DSL:* In our main DSL, the *Architecture Abstraction DSL*, the software engineers can semi-automatically create an abstraction of an

architectural component view based on design models or during architecture reconstruction. To address the rich concepts and variations of patterns, we propose to use architectural primitives [16] that can be leveraged by software engineers for pattern annotation during software architecture documentation and reconstruction. Architectural primitives are primitive abstractions at the architectural level (i.e. defined for components, connectors, and other architectural abstractions[1]) that can be found in realizations of multiple patterns.

- *Pattern Instance Documentation Tool:* Using the architectural primitive annotations, our approach provides a *Pattern Instance Documentation Tool* which automatically suggests possible pattern instances based on the architectural component view of a system and a pattern catalog.

- *Pattern Catalog DSL:* The pattern catalog contains a templates of the architectural patterns to be identified. It is customizable, reusable and integrates support for pattern variability. Our approach leads to a reduced search space for patterns, as we search for patterns only in the created architectural component view instead of the source code.

- *Pattern Instance DSL:* Identified pattern instances are documented using the *Pattern Instance DSL* which uses the artifacts defined in the *Architecture Abstraction DSL* and the *Pattern Catalog DSL* to permanently store pattern instance documentations.

We automatically generate traceability links between the architectural abstractions and the source code (more specifically, the automatically generated class models of the source code), the architectural abstractions and the selected pattern instances, and the pattern instances and the pattern catalog. When artifacts are changed, the traceability links are used to automatically check the consistency of all the artifacts. Automated consistency checking aids the software engineers during the incremental architecture documentation process, when new artifacts are identified and documented. For example, the system automatically detects when the pattern catalog is used to customize an existing pattern and these changes cause an existing instance of this pattern to be no longer valid. The consistency checks are used throughout the evolution of the documented system and report any occurring violations within seconds.

This article is structured as follows: In Section 2 we briefly explain architectural patterns and architectural primitives as our background. We give an overview of our approach in Section 3, and present it in detail in Section 4. In Section 5 we present three case studies of open source systems in which we have applied our approach to test its applicability. As it is crucial for our approach that it works smoothly in the

[1]Today, the component and connector view (or component view for short) of an architecture is a view that is often considered to contain the most significant architectural information [15]. Taylor et al. [17] define *components* as architectural entities that encapsulate a subset of a systems functionality and/or data. Each component has an explicitly defined interface that restricts access to the component's functionality and data as well as explicitly defined dependencies on its required execution context. They define a *connector* as an architectural building block that is tasked with effecting and regulating interactions among components.

working environment of the software designer during software design and development, we evaluate the execution time of our prototype in Section 6. In Section 7 we discuss lessons learned from the case studies and the performance evaluation as well as limitations of our approach. We compare to the related work in Section 8 and conclude in Section 9.

2. Background: Patterns and Architectural Primitives

A significant aspect of documenting software architectures is the representation of architectural patterns [4, 18] and the closely related architectural styles [19]. In general, a pattern is a problem-solution pair in a given context. A pattern does not only document 'how' a solution solves a problem but also 'why' it is solved, i.e., the rationale behind this particular solution. Architectural patterns help to document architectural design decisions, facilitate communication between stakeholders through a common vocabulary, and assist in analyzing the quality attributes of a software system.

Common approaches for modeling architectural patterns are Architecture Description Languages (ADLs) [20], the Unified Modeling Language [21], and formal or semi-formal approaches for the formalization of pattern specifications [22, 23]. As discussed in detail in our previous work [16], none of these approaches succeeds in effectively modeling architectural patterns, as they (1) are too limited in their abstractions to cover the rich concepts found in the patterns and (2) do not deal with the inherent variability of architectural patterns. To solve these problems we then proposed in [16] to remedy the problem of modeling architectural patterns through identifying and representing a number of *architectural primitives* that can act as the participants in the solution that patterns convey. We use the term 'primitive' because they are the fundamental modeling elements in representing a pattern, and they are the smallest units that make sense at the architectural level of abstraction (e.g. specialized components, connectors, ports, interfaces). Our approach relies on the assumption that architectural patterns contain a number of architectural primitives that are recurring participants in several other patterns [24]. These primitives are common among the different patterns even if their semantics demonstrate slight variations from pattern to pattern.

In our previous work, we provided modeling abstractions for each type of elicited architectural primitive [16]. In this work, we propose a semi-automatic architectural pattern identification and documentation approach based on the architectural primitives. The benefit of using this approach during architecture documentation and architecture reconstruction efforts is that the primitives can capture the rich concepts found in patterns as well as their inherent variability.

In the remainder of the paper we use the term primitive in two different contexts. First, we use the term primitives to annotate the architectural component view with the primitive information. In this context we use the primitives, as described above, as fundamental modeling elements. Throughout the the paper we refer to this as primitive annotations. The second context is the description of architectural pattern templates. In this context we use the term primitive as a parameterizable version of the definition above to describe the properties of a pattern participant.

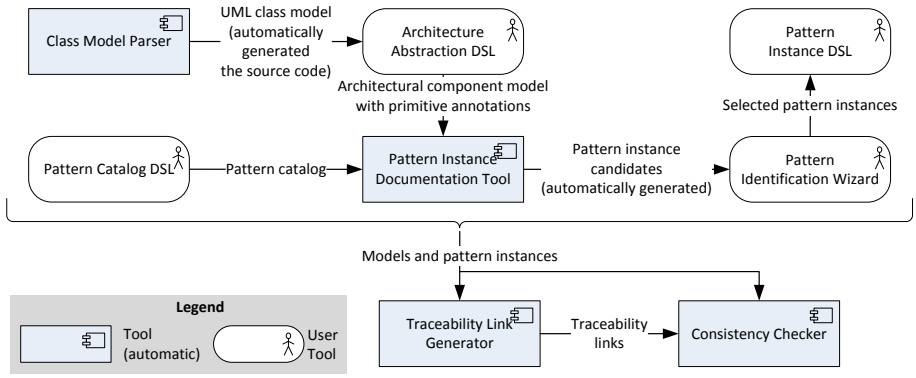

Figure 1: Overview of the approach

3. Approach Overview

Figure 1 shows the most important steps and tools in our approach. The central tool is the *Pattern Instance Documentation Tool*. Its goal is to document architectural pattern instances based on an architectural component and connectors view of a system that is annotated with architectural primitives.

The tool is semi-automatic, as it also receives manually edited inputs developed using the *Architecture Abstraction DSL*. In our previous work [25] we developed a basic version of this DSL that can be used to provide architecture abstraction specifications to incrementally create an architectural component view which abstracts over source code. In order to provide language independence we first automatically create a UML class model from the source code and then the software architect uses our DSL to manually create abstractions from this UML class model to create an architectural component view. This component view is then permanently stored as a UML components and connector model.

Traceability links between the class model and the architectural component view are automatically generated. Essentially, the DSL supports the specification of architectural components and connectors based on source code elements. In this work, we extended the *Architecture Abstraction DSL* (described in detail in Section 4.2) to also enable software architects to incrementally annotate the created components and connectors with architectural primitives during architecture documentation or reconstruction. The final input for the *Pattern Instance Documentation Tool* is a reusable pattern catalog. Usually the pattern catalog is defined once and can then be reused many times. The pattern catalog contains a number of templates for architectural patterns. For the task of creating and editing pattern catalogs we defined the *Pattern Catalog DSL* (details in Section 4.1) that uses architectural primitives as the basis for pattern descriptions.

Based on the information from the architectural components and the pattern catalog, the *Pattern Instance Documentation Tool* (see Section 4.3 for details) automatically computes which patterns from the pattern catalog can be instantiated based on the architectural component view specification. Using a *Pattern Identification Wizard*, all

pattern instance candidates are presented to the software architect. The software architect then chooses the candidates she wishes to document. The created pattern instance description, as well as the pattern instance candidates are instances of another DSL, the *Pattern Instance DSL* (see Section 4.4 for details). This DSL uses elements from the *Pattern Catalog DSL* and the *Architecture Abstraction DSL* to describe pattern instance documentations. It is used to review and edit the documented pattern instances and pattern instance candidates, or add missing information, e.g. within the *Pattern Identification Wizard*.

Once a pattern instance is documented, it is automatically checked for consistency throughout further iterations of architecture documentation effort and the future evolution of the system using the *Consistency Checker*. Whenever the source code, the architectural components, or the pattern catalog are changed, our tools test if any consistency rules, defined by the architectural primitives, are violated or if any relations and constraints that are defined in the pattern catalog are no longer satisfied. As shown in Figure 1 the Pattern-Architecture Traceability Generator automatically maintains traceability links between all elements that are shared between the *Pattern Catalog DSL*, the *Architecture Abstraction DSL*, and the *Pattern Instance DSL*. To maintain traceability with the source code, the Code-Architecture Traceability Generator automatically generates traceability links between the architectural components and the source code based on the *Architecture Abstraction DSL*.

For all manual steps in the approach, our tools provide a number of features that ease the life of the software architect. This includes code completion for the DSLs, auto-complete for names of existing artifacts, and automatic generation of traceability links. Furthermore the system provides detailed information about violated constraints and the violating artifacts.

4. Detailed Description of the Approach

Our approach introduces a reusable pattern catalog that contains architectural patterns, an architectural component view that is annotated with architectural primitives, and pattern instances based on the pattern catalog and the architectural component view. In this section we describe the concepts and languages used for realizing these different parts of our approach in more detail. To illustrate our approach, we use a running example based on the open source game *FreeCol* [26] which is a turn-based strategy game based on the old game Colonization, and similar to Civilization. The objective of the game is to create an independent nation.

The central tool in our approach is the *Pattern Instance Documentation Tool* as it is responsible for the computation of possible pattern candidates that are presented to the architect in order to document the architectural patterns found in the source code. It is described in Section 4.3; however, we describe the *Pattern Catalog* and the *Architecture Abstraction DSL* first, as their concepts are important for understanding the details of the *Pattern Instance Documentation Tool*.

4.1. Pattern Catalog

The basis of our approach are patterns that can be defined in a reusable pattern catalog. Figure 2 shows the most important parts of the Ecore meta-model for this *Pattern*

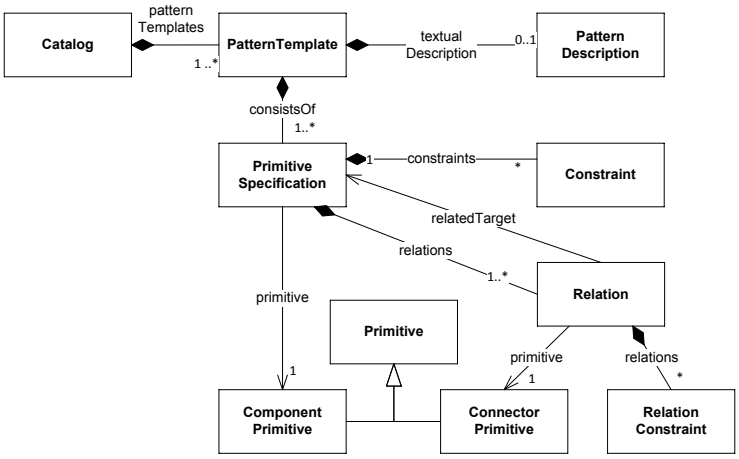

Figure 2: Excerpt of the Ecore model for the Pattern Catalog DSL

```
Pattern Template ModelViewController
      consists of:
      Model: Group (1)
      View: Component (1 .. *)
      connector to Model
      connector to Controller
      Controller: Group (1)
      connector to Model
      connector to View
      Description: "[...]"
```

Figure 3: Example showing the MVC pattern in the Pattern Catalog DSL

Catalog DSL[2]. It supports the definition of patterns based on architectural primitives.

Figure 3 shows an exemplary definition of the Model-View-Controller pattern as described by Fowler et al. [27]. This pattern aims at decoupling the presentation from the business logic and the data. It consists of three different parts: The Model that holds the data, the Controller that manipulates that data, and the View(s) that display the data. This pattern defines that the View and the Controller have a relationship with each other and also that both have a relationship to the Model. Please note the usage of the *Grouping* primitive that, in the context of the pattern template, defines the role *Controller* as a group of components that together fulfill this role.

A pattern specification (i.e. the `PatternTemplate` in the meta-model) consists of a textual description of the pattern and one to many `RoleDefinitions`. Each `RoleDefinition` describes one role and its relations to other roles that are part of the pattern. Thus an instance of `RoleDefinition` has a $Rolename$, a $ComponentPrimitive$, an arbitrary number of constraints, and a number of `Relation` objects. Each `Relation` requires a `ConnectorPrimitive` and a

[2]All our DSLs are implemented using Eclipse Xtext 2.3.1 utilizing Eclipse Xtend and Java for model-transformation and constraint checking (using the Eclipse Xtext validation framework).

target role and optionally holds a `RelationConstraint`. Currently three types of constraints are supported:

- *RangeConstraint*: this allows to define a lower bound (0 or greater) and an upper bound (1 to many - which is denoted by a *) for the occurrences of a role in a pattern instance description.
- *ExclusionConstraint*: this allows to define an exclusion: if a role is assigned in the pattern instance description one or more other roles must not be assigned.
- *RequiresConstraint*: this allows to define a requirement: if a role is assigned in the pattern instance description,one or more other roles have to be assigned as well.

This way `RoleDefinitions` can describe optional roles, by using a *RangeConstraint* with a lower bound of 0. Using the *RequiresConstraint* it is possible that a whole group of roles can be defined which all have to be assigned in the pattern instance documentation or none of the roles is assigned.

In Figure 3 three roles are shown: One with the name *Model* which specifies that a group of components belongs to this pattern and that the this group has no connector to elements fulfilling the roles *View* and *Controller*. One called *View* which specifies that one or more components are part of this pattern which have connectors to the *Model* and the *View*, and finally a group of components that are named *Controller* with connectors to *Model* and *View*. While the version of the MVC pattern that is shown in Figure 3 only allows a single controller there might exist versions of MVC that utilize multiple *Controllers*. In order to extend the pattern template to allow this variants, only the *RangeConstraint* that follows after *Controller* has to be modified from (1) to e.g. (1 .. *) to allow the unbounded assignment of *Controller*s in the pattern instance description.

4.2. Architecture Abstraction DSL

In our previous work [25] we introduced a DSL which we now extended and evolved into the *Architecture Abstraction DSL*. It is intended to describe a software system's architectural component view with its connectors and primitive annotations. This language supports the creation and maintenance of architectural component views during the software life cycle. It uses architectural abstraction specifications to map source code elements to architectural components using a number of different DSL clauses that group source code elements based on a systems structure (e.g. by associating a component with a UML Package, which in turn is based on a programming-language-level package), the relations between source code elements (e.g. by associating a component with all classes that implement a specific interface), and regular expressions on element names. In addition to the basic clauses it also defines the set operations union, intersect, and difference in order to define complex architectural abstraction specifications.

This DSL works takes a UML class model (as representation of the source code) as input and thus allows the support for different languages. While we only implemented a parser that automatically creates the UML class model from Java source code, other

```
Component ClientController
consists of
{
  Package ([...].client.control)
  or
  //[...]
}
is in Group ( Controllers )
connector to Server
implemented by
  //[...]
  or
  Class([...].server.FreeColServer)
connector to GUI
connector to Model
connector to ServerModel
//[...]
```

```
Component Interpreter
consists of
{
  Class (".*Interp")
  or
  {
  Package(root.frag.core)
  }
}
is a Shield for CommandGroup
is a Shield for Parser
indirection to CommandObjects
indirection to FileCommandObjects
connector to Parser
```

Figure 4: Example for an architectural abstraction for the `ClientController` component of the FreeCol system [26] as well as an example for an architectural abstraction for the `Interpreter` component of the Apache CXF [28] case study (Section 5.3).

languages can be supported by either implementing a parser or using existing tools that support the creation of UML class models from source code.

Once the architecture components and how they map to source code is defined by the architect using the DSL, a UML component view is automatically generated. During this automatic generation of the architectural component view, the *Architecture Abstraction DSL* also stores the traceability links between source code and architectural components in the generated component view as UML relations between the architecture components and the source code elements. It further supports the software architect by automatically checking the consistency of the architectural component view and the source code by checking the continued existence of source code elements as well as constraints on the use of source code elements in the architecture abstraction (e.g. source code elements that are mapped to multiple components) and consistency of relations that are defined on the architectural level with the relations between source code artifacts. This reduces the risk of new or further architecture erosion during the remainder of the system's life-cycle as invalid or outdated architectural documentation is similar to having no architecture documentation. However, having no architecture documentation would waste the effort of carefully designing the architecture of a system in the first place [15], as the architectural knowledge will leave the project with the person that designed the architecture.

We now build on this approach by extending the architectural component view and allowing the architect to (manually) annotate the abstractions for architectural components with architectural primitive information.

Table 1 provides an overview of the primitives that are used in the examples and cases in the article. In our prototype we have implemented all primitives defined in our previous work [16].

In the FreeCol example (Section 5.1) we identified 10 architectural components. Figure 4 shows the definition of one of these architectural components, `ClientController`, which contains the source code package `root.net.sf.free-`

Table 1: Overview of the defined primitives (excerpt)

Primitive Name	Annotation Target	DSL Keyword	Description
Component	(Component)	(Component)	Supertype for component primitives
Grouping	Component	Group	Component is part of a group of components
Layering	Component	Layering	Component belongs to a layer
Connector	(Connector)	connector to	Supertype for connector primitives
Callback	Connector	is callback for	Register a callback with another component
Indirection	Connector	indirection to	Indirection to another component
Aggregation Cascade	Connector	aggregates	Component aggregates other components
Composite Cascade	Connector	composition of	Component is a composite of other components
Virtual-Connector	Connector	virtually connected to	An indirect connection to another component
Shield	Connector	shield for	Prevents direct access to a set of other components

`col.client.control`. The `or`-statement in the code represents the union operation and indicates that more source code elements are contained in this component which we do not show here for brevity reasons.

In addition, this architectural component has been manually annotated with the *Grouping* primitive and been added to a group called `Controllers`. This definition also contains a set of connectors for this architectural component (also abbreviated). Similar to the ball and socket notation in UML 2, connectors in the *Architecture Abstraction DSL* have direction, as they pertain to an architectural component and can target either a single architectural component or a group of components that are annotated with the *Grouping* primitive. The connectors of the `ClientController` component shown in Figure 4 were automatically generated based on relations in the source code. This component, respectively the group it is annotated with, is a candidate for the role `Controller` in the Model-View-Controller pattern (Figure 3) as this architectural component has connectors to an architectural component `GUI` (which fulfills the constraints for the role View) and to the components `Model` and `ServerModel` which form a group called `ModelComponents`. This group fulfills the constraints for the role `Model`.

The components shown in Figure 4 also exemplify the annotation of architectural components with primitives. While the `ClientController` component is annotated with the *Grouping* primitives, which indicates that it is part of a group of components that belong together, the `Interpreter` component is annotated as a *Shield* that shields the `Parser` component and a *Grouping* called `CommandGroup` from direct access. The Xtext grammar of the *Architecture Abstraction DSL* can be found in Appendix A.

4.3. Pattern Instance Documentation Tool

In this section we first describe the *Pattern Instance Documentation Tool* of our approach. When architects intend to create a pattern instance description based on the pattern catalog, the architectural primitive annotations are used to automatically search for possible patterns that are then presented in an Eclipse Wizard. There the architect can select the pattern she deems appropriate. The wizard then creates a pattern instance description based on the selected pattern and prefills all values that can be automatically determined.

The *Pattern Instance Documentation Tool* first reads the architectural component view and the pattern catalog. The tool then checks for each pattern template in the pattern catalog if the pattern's constraints are satisfied in the architectural component view. The basic algorithm for this task is shown in Algorithm 1. It is invoked for every pattern template in the pattern catalog and returns a pattern candidate if the pattern's constraints can be satisfied.

Algorithm 1 Pattern Evaluation Algorithm

```
1:  procedure EVALUATEPATTERN(patternTemplate,model)
2:      candidate := new Candidate()
3:      for each primitive in patternTemplate do
4:          comps := GETANNOTATEDCOMPONENTS(model, primitive)
5:          if CHECKCOMPONENTCONSTRAINTS(primitive, comps) then
6:              UPDATECANDIDATE(candidate, primitive, comps)
7:          else
8:              return null
9:          end if
10:     end for
11:     for each primitive in candidate do
12:         if ¬CHECKRELATIONCONSTRAINTS(primitive, candidate) then
13:             return null
14:         end if
15:     end for
16:     return candidate
17: end procedure
```

In Algorithm 1, for each architectural primitive of the type `ComponentPrimitive` that is used in the pattern specification, the method GETANNOTATEDCOMPONENTS is used to find all architectural components that are annotated with the specified primitive. After this, the method CHECKCOMPONENTCONSTRAINTS is used to check if the constraints for the primitive (as specified in the pattern catalog) can be satisfied. When all component primitives of a pattern have been satisfied, the algorithm checks if the `ConnectorPrimitives` defined in the pattern template and all constraints defined for these `ConnectorPrimitives` are satisfied. If any constraint or primitive of a pattern template cannot be satisfied using the primitive annotations in the architectural component view, the evaluation of the pattern template is aborted. If all constraints are satisfied and all the pattern template's primitives exist as primitive annotations in the architectural model, the pattern template is accepted and a candidate is created. This pattern candidate is not a complete pattern instance description, as precomputed complete pattern instances would lead to a potentially huge number of pattern candidates. Each pattern candidate holds information about a single pattern template that can be found in the architecture and contains a map that holds, for each role, the architectural components that can be assigned to this role (based on the role's primitive and

the architectural component's primitive annotations). Algorithm 1 has a worst-case runtime complexity of $O(N \times M)$, i.e., in simplified form we can write $O(N^2)$. Because of the quadratic complexity of the algorithm we performed an evaluation of our approach's performance which is presented in Section 6.

In the Model-View-Controller(MVC) example (see Figure 3), this means that when creating an instance, it is necessary that the architecture abstraction specification matches the constraints for all `RoleSpecification`s that were defined in the pattern template. For example, to assign the `Model` role, at least one group primitive annotation needs to exist in the architecture abstraction, while any defined component is viable for the role `View`. To satisfy the connector primitives for the role `View` at least one architectural component is needed that has connectors to at least two distinct groups of components. Similar requirements are necessary for the role `Controller` which requires that a number of architectural components are grouped. To fulfill the role, this group needs to have a connector to the `Model` components and also needs to have at least one connector to the possible candidates for the `View` role. The pattern template for the MVC pattern also forbids a relation between the model and the view as well as the model and the controller. However it is not efficient to check for the absence of a relation during the search for possible pattern candidates. At this time no knowledge or limited knowledge (in case of roles that can only be fulfilled by one component or grouping) about the concrete pattern instance exists and it would be necessary to check all possible assignments (all possible pattern instances) for the absence of this relation.

The list of accepted pattern instance candidates is then presented to the software architect together with information about the pattern template that is the basis for the candidate. Once the software architect selects one or more pattern instance candidates, the *Pattern Instance Documentation Tool* tries to automatically assign components for the roles of the selected candidates. If not all the roles of a pattern instance candidate can be automatically assigned, the *Pattern Instance DSL* is used to present the unfinished pattern instance to the software architect and she needs to complete the pattern instance by selecting one of the viable architectural components for each unassigned role. Once all roles of a pattern instance are assigned, it is permanently stored for documentation and later use – again using the *Pattern Instance DSL*.

To support continuous consistency checking, all the checks that were performed in the *Pattern Instance Documentation Tool* during the creation of a pattern instance are performed for each selected pattern instance both during the following iterations of the architecture documentation and subsequent evolution of the system.

As discussed in Section 1 and 2, variability is inherent to all architectural patterns and their implementations. Two possible examples of pattern variations are models with pattern instances that contain additional architectural elements not described by the pattern or pattern instances where parts of the pattern have been omitted (an example for this case is shown and discussed in Section 5.3). The *Pattern Instance Documentation Tool* ignores additional architectural elements by default and is only influenced by additional architectural elements if the pattern template(s) in the pattern catalog explicitly forbid certain relations. If a pattern implementation omits parts that are specified in the corresponding pattern template in our pattern catalog, in order for the *Pattern Instance Documentation Tool* to identify the pattern, it is necessary to mark

```
Pattern Instance: ModelViewController
      Model : ModelComponents
      View : GUI
      Controller : Controllers
```

Figure 5: Example instance of the MVC-pattern for the program "FreeCol"[26]

the missing part of the pattern template as optional. We plan to support the search for incomplete implementations using an heuristic approach in the future.

4.4. Pattern Instances

For creating and persisting pattern instances we implemented the *Pattern Instance DSL* that references elements from the *Pattern Catalog DSL* and the *Architecture Abstraction DSL*. The pattern instances hold the information which architectural components relate to which parts (roles) of the pattern from the pattern catalog. In Figure 5 we show an example instance of the Model-View-Controller pattern that we identified in the architectural components of FreeCol. This instance uses the group `Controllers` (which holds the `ClientController` from Figure 4) for the role with the same name. It assigns the `Model` role to the `ModelComponents` group and the `View` role to the architectural component `GUI`.

For every pattern instance documentation, our *Pattern Instance DSL* expresses and permanently stores traceability links between the elements from the pattern instance description, the pattern template and its roles from the pattern catalog, as well as the architectural components from the architectural component view that are assigned to roles. This is done implicitly as the existing artifacts from the *Architecture Abstraction DSL* and the *Pattern Catalog DSL* are directly referenced when a pattern instance is documented. For the example shown in Figure 5, this means that `ModelViewController` is a reference to the pattern template from the pattern catalog and `Model`, `View`, and `Controller` are references to the roles that are defined in this pattern template. While `ModelComponents`, `GUI`, and `Controllers` are references to components or `Groupings` specified in the *Architecture Abstraction DSL*. All of these traceability links are navigable by the architect in the tool and allow a quick navigation between the different artifacts of our approach. E.g. the architect can navigate from the documented Model-View-Controller pattern instance in the *PatternInstanceDSL* to the underlying pattern template specified in the *PatternCatalogDSL* by Ctrl+clicking the pattern template name in the pattern instance documentation. This also works for roles and assigned components, where a Ctrl+click on the role `View` will also open the template for the Model-View-Controller pattern, while a Ctrl+click on the assigned component `GUI` will bring up the specification of this component in the architectural component view.

Based on the traceability links, our tool suite checks consistency of pattern instance descriptions with the architectural components, which in turn are consistency checked against the source code artifacts, as well as the underlying pattern templates. These checks also include the aformentioned checks whether a pattern template defines constraints on the relations of its participating roles (see Section 4.3). If the consistency checks detect a violation, an error is raised and the affected part of the pattern instance description is highlighted.

5. Case Studies

In this section we present three open source system case studies to better illustrate our approach and to study the practical applicability of our approach to do architecture reconstruction and documentation for existing, non-trivial software systems. Finally, the case studies are also used as a basis for the performance evaluations in Section 6. In the first case study we documented the architecture of the open source game FreeCol, which was partly presented as a running example before. In the second case study, Frag, we explain the documentation of the architecture of an open source programming language implementation – which we developed in our previous work – for a number of evolution steps. The documentation was performed by the first author who was not involved in the development of Frag. In the third case, we study the documentation of architectural patterns on an open source system with approximately 390.000 lines of source code in more than 2000 classes, namely Apache CXF.

5.1. Case Study: FreeCol

FreeCol is a turn-based open source multi-player game implemented in Java. The implementation uses a Client-Server architecture. For all games the clients connect to a server to play. While a local server is started for single-player games, a dedicated server can be used for multi-player games.

Through our architecture reconstruction and documentation effort we identified the architecture shown in Figure 8. We identified 10 components and their relationships. The client consists of a graphical user interface `ClientGUI` that displays the game based on a domain model (`Model`). All actions that are executed by the user are forwarded to the `ClientController` which updates the model accordingly and also notifies the server about the changes. For this purpose the `ClientNetworking` exposes a server API to the client. Calls to this API are then forwarded to the server using a message-based protocol that is implemented by the `Networking` component. On the server-side received messages are forwarded from the `ServerNetworking` to the input handler which is part of the `ServerController` where `MessageHandlers` are used for handling the input. These update the server's game state which is realized in the `ServerModel` and notify other players if necessary. This means that the API provided to the client actually does not have a direct counterpart on the server-side as the implementation of this fuctionality is distributed on the different `MessageHandlers`. On the client side, the `ClientController's InputHandler` reacts to messages that are received from the server and updates the GUI and model accordingly.

During the architecture documentation we decided to group the components that provide similar functionality for the client and the server. One obvious choice for grouping were the `ClientController` and `ServerController` which we grouped as `Controllers`. The second group we annotated was the `ModelGroup` which consists of the `Model` and the `ServerModel` component. Another set of components that together provide important functionality are all components concerned with networking which together provide the facilities for the communication between clients and the servers. Naturally we grouped them in a group called `Networking`.

During the source code study of this project we noticed that the `ClientNetworking` component as well as the `ServerNetworking` component both use the `Networking`

component to handle the input they receive. Thus we annotated the connector between `ClientNetworking` and `Networking` as well as the connector between `ServerNetworking` and `Networking` with the *Indirection* primitive.

Using these primitive annotations, the Pattern Instance Documentation Tool then proposed a number of patterns from the pattern catalog. Among these are the already mentioned Model-View-Controller (MVC) pattern, as well as the Broker [29], Application Controller, Page Controller, Proxy, Transform View, Template View, and Two Step View patterns (which are all discussed by Martin Fowler [27]).

After manual analysis of the architectural component model, we concluded that the MVC pattern matches the architecture's user-interface best, as alternatives like the Page Controller pattern are similar to the MVC pattern but do not match FreeCol's intended architecture. For the MVC pattern we selected the `ClientGUI` as `View`, the `Controllers` grouping as `Controller` and the `ModelGroup` grouping for the `Model` role. A very similar option would have been the Page Controller pattern. Our current pattern templates for the MVC and Page Controller patterns only differ in one relation between the role View and the role Controller which is forbidden in the Page Controller pattern and required in the MVC pattern. As already discussed in Section 4.3 the absence of relations can only be viably checked for existing pattern instances and the difference for these two patterns cannot be detected by the Pattern Instance Documentation Tool which reports both patterns as possible pattern candidates.

In order to demonstrate this, we also documented an instance of the Page Controller with the same role assignments that we used for the MVC pattern. However the consistency checker raises a constraint violation for the Model role once this pattern instance documentation is checked. This is shown in Figure 6. A similar constraint violation might occur when the application changes over time and new dependency between two components is implemented in the source code. This first leads to a constraint violation in the *Architecture Abstraction DSL*. If the architect then decides that an evolution/change of the architecture is necessary (rather then changing the source code), she adds the according connector to the architecture abstraction specification. This, in turn, may lead to a constraint violation like the one shown before. This works in a similar manner if dependencies between different components and thus their connectors are removed.

After the attempt to document an instance of the Page Controller pattern we also documented the Broker pattern [29]. This pattern's context are distributed objects and the transparent invocation of remote objects. For this purpose a client-side requestor and a server-side invoker are used that hide the implementation details of the network communication from the engineer using a marshaller. In order to provide type-system transparency the requestor usually is a proxy for the object that is invoked on the server.

Our pattern template which is based on the description of Völter et al. [29] was able to describe the implemented architecture. For this pattern instance the assignment of roles had to be done manually as the *Pattern Instance Documentation Tool* found more than one possible option for each role.

As the description of the Broker pattern is based on the *Component* primitive, the architect has to select between all of FreeCol's components when assigning the first role. After the role `Client` was assigned to the `ClientController` component, the tool suggested the 3 components `Model`, `ClientGUI`, `ClientNetworking` for

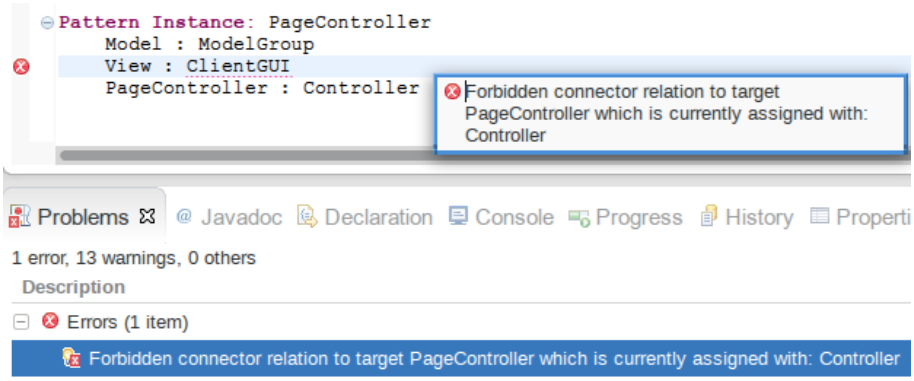

Figure 6: FreeCol case study: Page controller pattern instance with constraint violation[26]

```
Pattern Template Broker
consists of:
Client: Component
connector to ClientProxy
ClientProxy: Component
connector to Transport
Transport: Component
ServerProxy: Component
connector to Transport
Server: Component
connector to ServerProxy
```

```
Pattern Instance: Broker
Client: ClientController
ClientProxy: ClientNetworking
Transport: Networking
ServerProxy: ServerNetworking
Server: ServerController
```

Figure 7: FreeCol case study: Broker pattern template and Broker pattern instance description

the `ClientProxy` role. After the selection of the `ClientNetworking` component as `ClientProxy`, our tool automatically suggested the `Networking` component for the `Transport` role. For the `ServerProxy` role, the tool provided a choice between the `ClientNetworking` and the `ServerNetworking` component. At the current point of time, however our tool does not use other means than structural information and thus cannot automatically select the `ServerNetworking`. The last remaining `Server` role was automatically assigned to the `ServerController` component. This results in the following assignment for the documented pattern instance: `ClientController` as Client, `ClientNetworking` as ClientProxy, `Networking` for the Transport role, `ServerNetworking` as ServerProxy, and `ServerController` as Server. Figure 7 shows the template for the Broker pattern in the Pattern Catalog and the pattern instance documentation for the implementation found in FreeCol.

This case study shows the applicability of our tool for a medium sized system with about 100k lines of source code. The annotation of components with architectural primitives proved to be a straightforward task in this case, as the chosen annotations came quite naturally during the source code study. After the creation of the architectural component view, the documentation of the two architectural patterns required only little effort. Both were automatically suggested by the *Pattern Instance Documentation Tool*, and, while there were no alternatives found for the Broker pattern, a number of alternatives were suggested for the MVC pattern. The case study also shows that

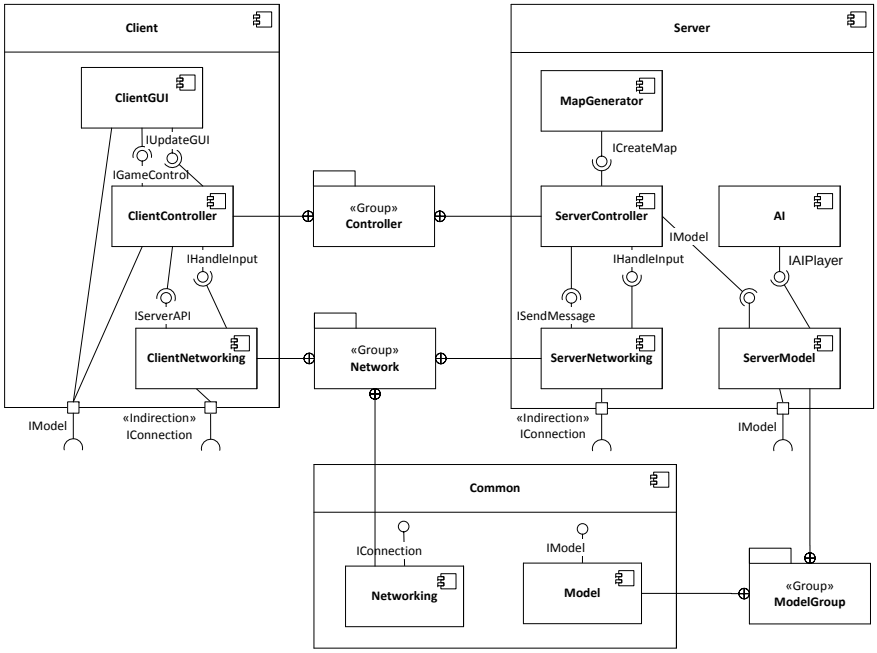

Figure 8: FreeCol architecture overview [26]

the distinction of similar user interface patterns like Model-View-Controller or Page Controller on this level of abstraction can be a challenging task and requires human judgment. However, our approach aids the software designer in selecting the appropriate patterns by providing traceability links from the architectural information to the source code elements.

5.2. Case Study: Frag

Frag [30] is a dynamic programming language that is designed to be tailorable, support the creation of DSLs, support Model-driven Development, and the Frag interpreter is embeddable in Java, in which Frag also is written. We applied our approach to document Frag's architecture. We started by creating an architectural component view and pattern instance documentations for Frag version **0.6**. In this first iteration of our architecture documentation effort, we identified 4 architectural components and annotated these with primitive information.

As Frag is an interpreted language, the most important architectural component is the `Interp` component. It provides two interfaces. On the one hand, it allows the embedding of Frag in any Java program, and, on the other hand, it provides the functionality to execute the commands that were given as input via the components `Client` and `Shell` or via the `IEmbeddingFrag` interface. As the `Interp` uses different command objects to execute the received commands, we annotated the involved connectors with *Indirection* primitives. In addition the connectors to `Interp`'s provided interfaces

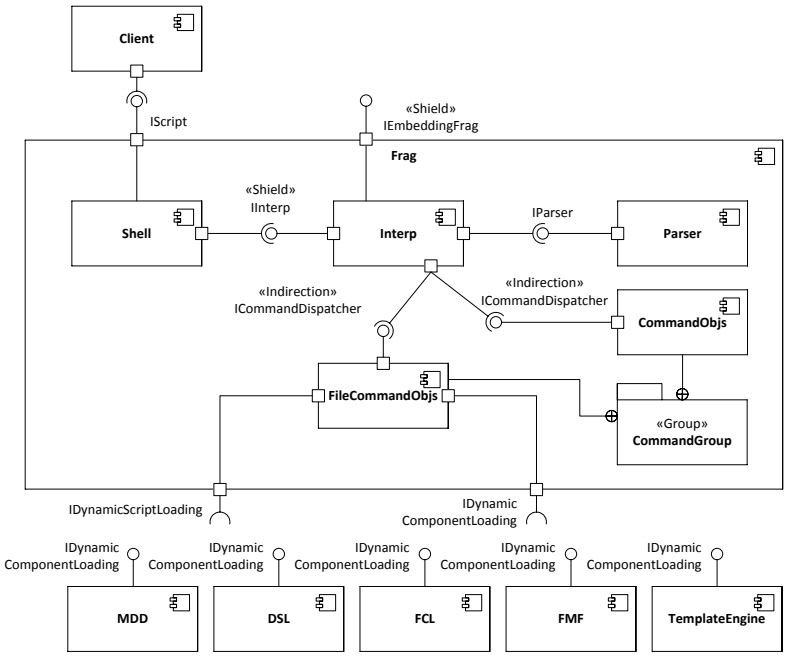

Figure 9: Architectural component view for Frag 0.91

are annotated with the *Shield* primitive as the `Interp` component shields the access to the `Parser` and `CommandObjs` components.

Pattern Documentation. On this architectural component view we used our *Pattern Instance Documentation Tool* to identify all pattern candidates. Based on our pattern catalog, our tool provided the following candidates: Facade [18], Indirection Layer [18], and Interpreter [18]. The Facade pattern simplifies the access to a complex subsystem and decouples the client code from the actual implementation of the subsystem. While Facade is often used as design pattern, it can also be used on architectural level as sometimes access to a whole subsystem can be provided by an architectural component that provides a simplified interface for this subsystem to the rest of the application.

An Indirection Layer differs from the Facade as it is intended to hide the actual implementation of a subsystem and should not be bypassed, while a Facade still allows direct access to a subsystem. Similar to a Facade, it is possible that an Indirection Layer [18] holds additional logic or performs additional tasks. The Interpreter pattern defines a class-based representation for a grammar along with an interpreter to interpret the language defined by the grammar [31]. Although the patterns have different intents, from the perspective of the *Pattern Instance Documentation Tool*, all three candidates are plausible for the architectural model of Frag because the structural descriptions of the patterns (see Figure 10) are similar.

For both patterns in Figure 10 a number of possible variants for the description of the patterns exist. For the Interpreter pattern one variant would be that the Ex-

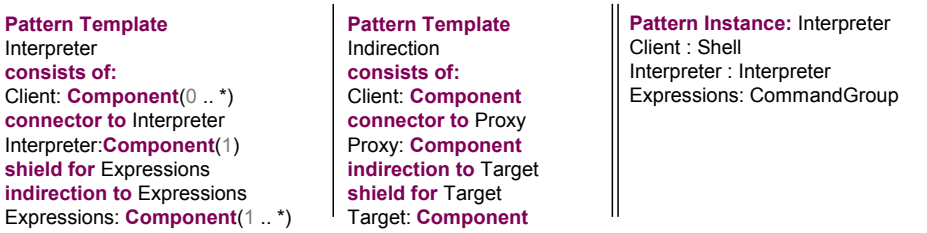

Figure 10: Pattern templates for the Interpreter and Indirection patterns as well as the pattern instance of the Interpreter pattern in the Frag example

pressions that implement the language are grouped into one architectural component instead of a group of components. Another variant could be that the Expressions are implemented in the architectural component that fulfills the `Interpreter` role and the `Expressions` is omitted. In the same way, variants of the Indirection Pattern are possible where e.g. the *Shield* might not be necessary.

The pattern instance we selected from these candidates is the `Interpreter` pattern because of multiple reasons. The first indication is the name of the reconstructed `Interpreter` component, and secondly this architectural component dispatches to the component, containing command objects, and this execution of commands matches the Interpreter pattern better than it matches the Facade or Indirection pattern.

After this selection, the *Pattern Instance Documentation Tool* automatically uses the `Interpreter` component for the role `Interpreter` from the pattern template and the `Shell` component as the role `Client`. The role `Expressions` had to be assigned manually as more then one possible assignment exists in our architectural component view. For the role `Expressions` we did select the component `CommandObjects`.

Architecture Evolution. In order to study consistency checking during architecture evolution, we then updated Frag's source code to version **0.7** and let the *Consistency Checker* test for inconsistencies. In a first step, our *Consistency Checker* reported a number of classes that existed in the source code but were not considered in the architectural abstraction as well as a package that was referenced in the architecture abstraction specification but no longer existed (as it had been renamed). The renamed package resulted in an update in the architecture specification where the reference to the package was changed accordingly. After a source code study of these new classes, we introduced four additional components to the architecture abstraction called `FileCommandObjects`, `MDD`, `DSL`, and `FCL`. Furthermore we added a connector from `Interpreter` to `FileCommandObjects`, one between `FileCommandObjects` each of the other new components. In addition our source code study had revealed that the *Interpreter* acted as a *Shield* for the `FileCommandObjects` and that the `Interpreter` now also used this component to execute commands. While the `FileCommandObjects` component utilized the other new components using dynamic loading. This is why we added a *Shield* and an *Indirection* primitive annotation for the `FileCommandObjects` component.

After these changes the *Consistency Checker* reported that the new components were not part of any documented pattern instances and suggested the

Table 2: The number of traceability links created or deleted by the *Traceabiliy Link Generator* during each evolution step.

Version change	Number of created traceability links	Number of deleted traceability links
Frag 0.6 → 0.7	122	113
Frag 0.7 → 0.8	53	22
Frag 0.8 → 0.9	59	91

`FileCommandObjects` as another participant of the documented `Interpreter` pattern. Specifically the component was suggested to be also assigned to the `Expressions` role of the documented `Interpreter`.

We then continued this process until we reached Frag's current version 0.91. In version **0.8** the *Consistency Checker* again reported new classes, which led to another two new components in the architecture abstraction specification that are connected to the `FileCommandObjects` component, as well as a package that had been renamed. This required an update to the architecture abstraction specification. After updating the source code from version 0.8 to version **0.91** the consistency checker did not report any inconsistencies as all new classes were already covered by the architecture specification and thus no changes to the architecture were necessary.

Traceability. Whenever we changed the architecture abstraction specification, our the *Tracebility Link Generator* recalculated all traceability links. As shown in Table 2, for each evolution step (version change) the generator created and removed a substantial number of traceability links. This is also true for the evolution step from Frag version 0.8 to Frag version 0.91 where the changes to the source code resulted in 59 new and 91 deleted traceability links although no changes to the architecture abstraction specification had occurred. Keeping these traceability links manually up-to-date requires a substantial effort by the software architect or developer.

Summary. During multiple iterations we identified a total number of 10 architectural components and their connectors. The final architectural component view for Frag is shown in Figure 9.

This case illustrates that annotation with primitives can easily be done while documenting a systems architecture using our DSL-based approach. It shows how the consistency checks support the architect during the future evolution of a system once a it's architecture has been documented using our approach.

5.3. Case Study: Apache CXF

Apache CXF is an open source Web services framework that is developed in Java that supports a wide variety of protocols like e.g. SOAP and RESTful HTTP. We used the architecture overview that is available at the CXF web-site[3] as a basis, and incrementally improved and annotated the architectural component view.

[3] `http://cxf.apache.org/docs/cxf-architecture.html`

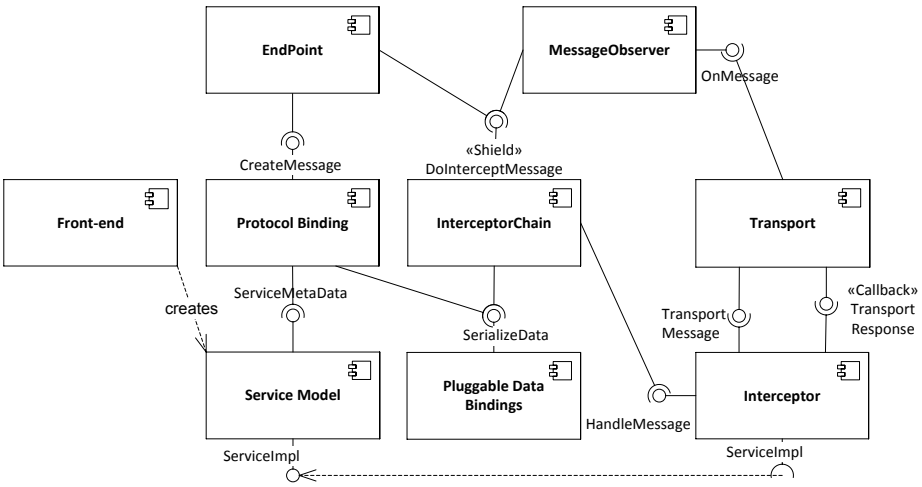

Figure 11: Apache CXF architecture overview [28]

The architecture of CXF 2.4.3 is built around an interceptor chain that is configured to handle all incoming request and outgoing responses on the server-side and on the client-side. As already mentioned it supports different protocols by allowing different protocol bindings and uses different transports to send and receive messages. This means that whenever Apache CXF receives the request to call a specific service the interceptor chain is configured to contain the necessary interceptors for the protocol and so on. The invocation is forwarded through the chain until finally one interceptor in the chain calls the service that was discovered using the service model and then another interceptor uses a conduit on the transport to send the result of the service call to the requesting client as a reply.

During our architecture documentation, we identified a number of cases where components realized characteristics of architectural primitives and annotated the components accordingly.

In particular, the `InterceptorChain` is the only component that accesses the interceptors in the architectural component `Interceptor` and thus was annotated as a *Shield* for the `Interceptor` component. Here, we combined all Interceptors that are implemented in Apache CXF into the `Interceptor` component. Another possibility (i.e., possible variant of the architectural primitive model) would have been to treat each Interceptor as an individual component and use the *Grouping* primitive annotation to combine them. The resulting architectural component view is shown in Figure 11.

During the source code study, we also found that the Transport Response connector between the `Interceptor` and the `Transport` components is actually a callback and thus was annotated with the *Callback* primitive.

After we finished the documentation and annotation of the abstracted components with architectural primitive information, our *Pattern Instance Documentation Tool* automatically found possible candidates for the patterns Facade [12] and Interceptor [18, 32]. The Interceptor pattern's [18, 32] intent is to increase a system's flexibil-

ity and extensibility by allowing to transparently updating the services offered by a framework [18]. Applications can register interceptors by adding or removing interceptors to or from a dispatcher at runtime. The dispatcher then notifies the interceptors of events that are sent by the framework's core.

The Facade pattern actually is a design pattern that is used on an architectural level in different variants throughout the literature [18, 27, 33]. Selecting the Facade pattern, defined as a Client that uses a Shield to access a specific group of architectural components, is an option in this case, as the `Interceptor` component is actually hidden in a Facade-style. However, in this case, the more sophisticated Interceptor pattern seemed the better choice, as (1) indicated by the component names and (2) obvious after closely inspecting the intent of the respective components and classes. While creating the new pattern instance, we discovered that our description of the Interceptor pattern was too narrow. Our description consisted of a `Caller` component, a `ChainHandler`, one or more `Interceptor` components that are "shielded" by the `ChainHandler` and a `Callee`. When creating the pattern instance documentation, we started to assign the roles as follows: the `Interceptor` component to the `Interceptors` role, the `InterceptorChain` as *ChainHandler* role, and `EndPoint` as `Caller`. However after the assignment of the `ChainHandler` role, the automatic consistency checks detected a constraint violation with respect to the pattern template. The role `Callee` was still unassigned. However when looking at the architectural component view, no suitable candidate for the `Callee` role could be identified and thus additional time was invested in studying the implementation of the pattern in the source code. Beside the fact that This manual study, during which we made extensive use of the automatically generated traceability links, came to the result that Apache CXF's implementation of the `Interceptor` pattern does not include a `Callee` as the Interceptor Chain handles all the logic. So, we modified our template of the pattern by allowing zero to one `Callee` instead of exactly one.

In the implementation of Apache CXF, the client and the server both use an instance of the interceptor chain, although they configure different interceptors. This required us to assign both the `EndPoint` and the `MessageObserver` for the `Caller` role which resulted in another constraint violation as our pattern template that was based on the literature only allowed to assign one `Caller`. In order to account for this, we updated the pattern template to allow multiple callers in the pattern description by adding a multiplicity of `(1..*)` to the *Caller* role. Once the description was updated, we assigned the roles like this: The architectural components `Endpoint` and `MessageObserver` as `Callers`, the `InterceptorChain` was assigned the role of `ChainHandler` and the `Interceptor` component was assigned to the role with the same name. The role `Callee` was not assigned.

We expected that, using our approach, we would also find an instance of the Broker pattern [29]. However, we did not identify this pattern – neither using the Pattern Instance Documentation Tool nor by manual identification of participating architectural components. The cause is that the structural aspect of the Broker pattern, as documented so far in our pattern catalog, does not exist in Apache CXF because it does not use a static setup for handling requests and responses. Apache CXF uses its `InterceptorChain` on the client and on the server and configures the `Interceptors` accordingly. So while it shows the behavior of a Broker pattern this

is not reflected in its structure. As a result, we extended our pattern catalog further to also include this variant of the Broker, and finally selected both Broker and Interceptor using the *Pattern Instance Documentation Tool*. We note that the Broker pattern could potentially be identified more precisely, if we would also include behavior information. We will investigate the problem how to integrate behavior information with our approach in future works.

This case illustrates, for an open source system with a substantial number of classes, how the annotation with architectural primitives can be performed during the documentation of a systems architecture using our DSL-based approach. It also illustrates how the pattern catalog is incrementally improved and extended (here one improvement of the Interceptor pattern description and one new variant of the Broker pattern have been explained), to show how our approach can deal with pattern variability. In our future work, we plan to extend our approach with a distributed pattern repository, in which all user updates are stored, so that different users of our approach can benefit from pattern variants documented by others.

6. Performance Evaluation of the Pattern Instance Documentation Tool

For the practical applicability of our approach it is crucial that it works smoothly in the working environment of the software designer during software design and development. To test the applicability of our approach in practice we measured the time it takes our *Pattern Instance Documentation Tool* to find pattern instances for our case studies and in 5 larger (with respect to number of components) synthetic component models. For the synthetic component models we used the basic structure of the Apache CXF case study and created multiples of the number of components from the case with varying component names and some additional random interconnections. Measurements indicate that the time required to search for pattern candidates increases with the size of the component model. However our prototype is able to search for patterns in synthetic component models with more than 350 components in reasonable time while the component models in our case studies do not exceed the number of 12 components.

For all the examples we used the same pattern catalog and primitives (explained above) which contained templates for 15 architectural patterns from the literature [29, 27, 4, 18] which includes architecture patterns like MVC, Broker, ApplicationController, PageController, Interpreter, Layers, and WrapperFacade. We measured the time it takes to run the *Pattern Instance Documentation Tool* a thousand times for each of the case studies and synthetic models. To obtain realistic results in a software developer environment, the measurement was performed on a developer notebook (Intel i7 L620, 8 Gb RAM) running Fedora 20 using Eclipse Kepler 4.3, Oracle Java 7. In Table 3 we present the number of architectural components, the standard deviation σ, the average, and median values of the execution time for all test cases. We do not report minimal and maximal values as the standard deviation is small compared to means and medians.

Our results indicate that our approach is usable even for larger component models (usually component models have not more than 5-20 components) on an average developer machine. We did not test varying the sizes of the pattern catalog, as our *Pattern Instance Recovery Tool* iterates through the pattern catalog with a loop, calling Algo-

Table 3: Results of the performance measurements for the case studies and larger synthetic models (in milliseconds, each executed 1000 times)

Example	number of arch. components	σ	Average	Median
Frag	10	0.84	1.37	1
FreeCol	10	1.52	7.21	7
Apache CXF	12	1.02	8.96	9
Synthetic model 1	24	1.73	11.23	11
Synthetic model 2	48	2.64	11.18	11
Synthetic model 3	96	4.58	19.46	19
Synthetic model 4	192	4.88	34.50	33
Synthetic model 5	384	5.42	70.59	69
Synthetic model 6	768	93.38	331.66	321
Synthetic model 7	1536	148.14	1334.02	1302

rithm 1 for each loop iteration, meaning that the performance of this complete loop is directly proportional to the size of the pattern catalog.

7. Discussion

In this section we briefly discuss the lessons learned from the three case studies and the performance evaluation.

7.1. Lessons learned from the case studies

The three case studies show the applicability of our approach for three different types of software and different kinds of architectures. While FreeCol [26] is a multiplayer game with a graphical user interface and a client server architecture, Apache CXF [28] is a web-service framework with an architecture based on an interceptor chain. The third case study, Frag [30], is a dynamic scripting language implemented in Java that is built around an Interpreter architecture. The cases also vary in size as Frag has about 10.000 lines of code, while FreeCol is much bigger and has about 100.000 lines of code. The biggest system in the example cases is Apache CXF which has about 350.000 lines of code. While all three case studies have about the same number of architectural components in the component view they differ in the abstraction level on which architectural patterns are documented.

Before applying our approach to the three case studies, we manually created our initial pattern catalog based on architecture patterns from the literature. The process of creating a pattern template for a documented pattern consisted of the time necessary to understand the pattern (which is always necessary) and the effort to describe the pattern's structure using our *Pattern Catalog DSL*. In our experience, the description of the pattern with the *Pattern Catalog DSL* required about fifteen minutes per pattern. Ideally the pattern catalog is publicly available and maintained by the community in order to be reused and adapted by individual users.

Table 4: Number of source code artifacts (classes and interfaces) compared to architecture artifacts (components and connectors) which need to be considered during architectural pattern identification.

Arch. Pattern	Source code artifacts	Architecture level artifacts
MVC (FreeCol)	459	10 components and 13 connectors
Broker (FreeCol)	169	10 components and 13 connectors
Interceptor (Apache CXF)	2340	12 components and 19 connectors
Interpreter (Frag)	174	10 components and 10 connectors

During our case studies we had to evolve and improve our pattern catalog twice, giving us the opportunity to test the effort required for adapting or creating a new pattern (variant). The necessary effort to create a new pattern variant consists of selecting the original template and then modifying the new variant which in total required not more than a few minutes for the Broker variant. A simple relaxing of constraints as discussed for the Interpreter pattern required only a single change in the pattern catalog without any need for closing and restarting our tool, as changes to the pattern catalog are automatically propagated.

For us the source code studies of the example systems naturally led to the architectural primitive annotations we reported and no additional effort was required to specifically search for possible options to add primitive information. However, somebody without knowledge about architectural primitives probably requires initial effort to learn about the architectural primitives and their functions before the annotation of architectural primitives during architecture reconstruction and documentation. Compared to the manual identification of architectural patterns, our approach also requires the architect to execute a source code study in order to create an architectural component view and thus could potentially require the same amount of effort. However, the case studies (Section 5) showed that the identification of architectural components in source code can be done in an iterative fashion and does not require the architect to study the complete source code at once, while manually identifying architectural patterns in the source code often requires to study a huge amount of classes at once and hence is more challenging than our approach where the architectural patterns are identified in the architectural component view. This is also shown in the case studies where the number of source code artifacts that are related to the implementation of the architectural patterns ranges from 169 to 2340 (see Table 4).

Once the architectural components and patterns are documented, our approach provides the software architect with automatically generated traceability links and automatic consistency checking. As already discussed in Section 5.2 and shown in Table 2, even for the Frag case study the number of traceability links that needed to be updated with each new version was between 75 and 235. Manually creating and updating these traceability links would be a tedious and error prone task.

Table 4 shows the discrepancy in the number of elements that have to be considered when identifying architectural patterns on the level of source code and on the level of architectural components for our case studies.

Once the architectural components were documented and annotated with architectural primitive information, the documentation of architectural patterns based on this primitive information required only two manual steps: The selection of suitable pattern

candidates from the list of pattern candidates that were automatically provided by the Pattern Instance Documentation Tool and then assignment of all the pattern roles which could not be automatically assigned by the tool.

Regarding our research questions we can draw the following result:

RQ1 Regarding the semi-automatic identification of patterns during architecture reconstruction, we could show the feasibility of our approach through the implementation of our prototype and through our case studies which exemplify the documentation of architectural patterns during architecture reconstruction. However, this approach requires a reusable pattern catalog as its basis. While we created an initial pattern catalog based on the literature, the creation and maintenance of this catalog require some effort which might hinder the adoption of this approach. Thus tools for sharing and maintaining pattern catalogs are required to ease the adoption of this approach.

RQ2 Regarding the maintenance of architectural patterns during the evolution of a reconstructed architecture, we can state that our approach supports further architecture evolution once patterns are documented. This is exemplified in Case Study 5.2. As discussed in the limitations below, while our personal experience from the execution of the case studies indicates a reduced effort for documenting and maintaining architectural patterns during the evolution of a system, we currently cannot not provide scientific evidence that our approach reduces the required effort. We will conduct a controlled experiment to investigate this topic further in the future.

RQ3 In our case studies, we have shown the applicability of the approach for three different types existing real-life systems with different project sizes. Therefore it is likely that our approach can be generalized to other similar cases. However, the component models in our case studies all have a similar number of components. As discussed in the limitations below, a significantly higher number of components in the component models might lead to a high number of pattern candidates and thus diminished benefit for the users of the approach during the identification of patterns.

RQ4 Regarding the efficiency of the actual pattern instance matching algorithms, the performance evaluation in Section 6 shows that our prototype is sufficiently efficient to be used for architectural pattern identification on common developer computers for artificial component models with 300 and more architectural components and thus should be efficient for day-to-day use.

RQ5 With respect to the adequacy of the primivites and the adaptable pattern catalog to handle the variability of architectural patterns, we can state that our case studies show that the concept of primitives can be applied to document architectural patterns and, as already discussed in Section 5.3, that only a small effort was necessary for evolving our pattern catalog during the case studies. Furthermore, during the creation of the initial pattern catalog and throughout the case studies, we were able to express all patterns based on the primitives in our Pattern Catalog DSL.

The case studies of FreeCol and Apache CXF showed a limitation of the current approach which is based on structural information only. For some patterns like the Page Controller and MVC patterns, which only differ in one relation, that is required in the MVC pattern and forbidden in the Page Controller pattern, it is hard to distinguish the structural differences during the computation of pattern candidates. This is because the forbidden relations cannot be taken into account during the creation of pattern candidates; however, later in our tool chain, our consistency checks for the documented pattern instances would have detected the constraint violation. The problem of distinguishing structurally similar patterns will be improved in our future work by also considering behavioral models of architectural patterns.

The pattern templates in our initial pattern catalog are based on the pattern descriptions from various sources in the literature (e.g., [29, 27, 4, 18]). Sometimes the templates from the literature are appropriate, and sometimes manual modifications are required. For example, during our case studies we could directly use the templates of the MVC, Broker, and Interpreter patterns we created based on the available literature, while it was necessary to modify the template for the Interceptor pattern and to create a new variant of the Broker pattern that was suitable to describe its implementation in Apache CXF.

While our approach supports the software architect during the source code study with information on which source artifacts in the architecture abstraction specification are not yet covered and provides traceability links for source artifacts that are already covered, it does work semi-automatically and still requires the software designer to perform a source code study. While automatic approaches try to free to developer of this burden, they usually discover a substantial number of false positives that have to be checked and corrected by the software designer. This leads to the necessity of doing a source code study anyway. Our approach on the other hand focuses on supporting the architect during the documentation of the architecture with tool support for the documentation as well as partial automation of the documentation steps. This includes the automatic generation of connectors between architectural components based on the relations between the components in the source code as well as the automatic suggestion of architectural patterns that match the structure implemented in the documented system. While these suggestions also contain false positives, they do not consist of complete instances (e.g. a suggestion that holds all possible instances of the MVC pattern), but only a list of patterns and if the software architect selects a pattern for documentation, our prototype of the approach supports the software architect during the assignment of roles. This includes providing a list of possible role-assignments based on the already existing role-assignments as well as automatically assigning roles where possible (see Section 5.1 for examples).

Later on, during the evolution of a system, automatic approaches usually have to start from scratch, while an architecture that is documented using our approach, is automatically checked against the system's source code without any additional effort. While we cannot provide any quantitative data on the benefits of consistency checks, they have been proposed and used in different contexts for almost 20 years now [34, 35, 36]. In addition our approach provides automatically generated traceability links for the documented architecture. In a recent controlled experiment [37], traceability links between architectural component models and the source code have proven to be

highly beneficial for architecture understanding.

While the main use case of our approach are systems without existing architecture documentation, it can also be used to formally document other existing (informal) architecture documentation to check if all consistency constraints are fulfilled. A combination with other complimentary forms of architecture documentation like architectural decisions is possible as well.

7.2. Threats to validity

In addition to the limitations already discussed above as lessons learned from our case studies, our case studies and performance evaluations have the following main threats to validity:

- The case study might not be representative to show the general applicability of the approach. As already discussed in the lessons learned, we tried to mitigate this threat by choosing cases from different application areas with varying sizes.
- As our approach requires input about the architectural primitives from the software designers, our results strongly depend on the quality of information provided by the software designers. We strive to improve the input quality by providing tools that support the software designer during the software architecture documentation, but ultimately our approach relies on the assumption that it is substantially easier to model with or detect the architectural primitives than patterns. Our experience so far shows that this assumption is justified.
- At this point we did not perform an evaluation of the applicability of the approach with other users, however we present three extensive case studies that showcase the applicability of the approach for three already existing systems.
- The synthetic models used in the performance evaluation might not be representative. We tried to mitigate this threat by using a real world model as a basis and created multiples of the case with custom component names and additional randomly created interconnections. In addition, we also measured the performance for our three case studies which are existing, realistic systems of varying size and which yield similar results.
- The pattern catalog used in the performance evaluation might not be representative. In order to reduce this risk, we used the pattern catalog that we initial created based on architectural patterns from the literature and that included all changes that were discussed in the example cases. As Algorithm 1 is executed for each pattern template, the execution time has a direct relation to the size of the pattern catalog.
- The effort necessary to create and maintain a useful pattern catalog might be large enough to hamper the usage of our approach. We tried to mitigate this risk by making the manipulation of the pattern catalog easy. The pattern catalog DSL is straightforward to use and the only required knowledge is the same as the one required to use our approach – knowledge about pattern primitives. However, we cannot fully eliminate this risk and other approaches faced this kind of problem before.

- The possiblity remains that the benefits do not outway the costs in the real world. A comparison in a controlled environment would be needed to contrast our appraoch's effort and the effort required to manually perform the same tasks. While we intend to perform this comparison as a controlled experiment in the future, our personal qualitative experience from performing the case studies shows an initial effort that is slightly higher than purely manual documentation for documenting the architectural patterns and a significantly reduced effort in maintaining the documented patterns during architecture evolution through the automatic consistency checking and the automatically maintained traceability links. This initial higher effort stems from the requirement to gain an understanding of the primitives as well as the need to learn to use our three DSLs. However this additional effort is only required when applying the approach for the first time. For our controlled experiment, we will follow the guidelines proposed by Kitchenham and Wohlin [38, 39]. The planned experiment will consist of a control group and a treatment group. While the control group will perform at least one architectural recovery and at least one architectural evolution task manually (using only an IDE but not our approach), the treatment group will perform the same tasks using our approach in addition to using an IDE.

- We applied the approach in three case studies that describe systems of different sizes. However, their architectural component models all consist of about ten components. A thread to validity of this approach is that this approach might not scale to systems which have a much higher number of architectural components, contain more architecture patterns, or are much larger in terms of source code size (like large-scale industrial systems). In this article, we only studied this scalability aspect in terms of the performance measurements for our Algorithm 1 which indicate acceptable performance for synthetic architectural component models with up to 350 components. However the threat to validity that a huge number of components, combined with a high number of primitive annotations, might lead to too many possible patterns and thus leads to a diminished benefit for the users of the approach, remains.

8. Related work

In this section we compare our work to related approaches that focus on modeling or detecting patterns or other approaches that utilize patterns during software architecture reconstruction or software architecture evolution. In Table 5 we give an overview of the related work discussed in this section and also provide a short comparison of this related work. Most of the related works focus on automatic design pattern identification while only a limited number focuses on finding architectural elements. As already discussed in Section 1 automatic approaches are limited by a high number of false positives while existing semiautomatic approaches focus either on a specific pattern [40] or on design patterns only [41]. In contrast, our approach focuses on architecture documentation and evolution of architectural patterns. It provides support for pattern variants and does not have the drawback of finding many false positives as it is semiautomatic and requires the software architect to annotate the architecture model with

architectural primitive information. In Subsection 8.1 we discuss approaches with a focus on architectural patterns, while Subsection 8.2 discusses approaches focusing on design patterns.

Table 5: Comparison of related approaches

Approach	Pattern types	Method	Pattern variants support	Automation	Focus
Medividovic et al. [42]	Architectural styles	Model comparison	None	Manual	Stemming architecture erosion
Yan et al. [43]	Architectural styles	State-machine	Design	Automatic	Stemming architecture erosion
Scaniello et al. [40]	Layers pattern	Link analysis	Implementation	Semiautomatic	Pattern identification
Sartipi [44]	Architectural patterns	Graph transformation and AQL queries	A* heuristic matching	Automatic	Architecture reconstruction
Harris et al. [45]	Style-library	Source code queries	Implementation	Automatic	Pattern identification
Lungu et al. [46]	Packaging patterns	Custom script	None	Semiautomatic	Architecture reconstruction
Paakki et al. [47]	Architectural and Design patterns	CSP	None	Automatic	Quality Assessment
Di Penta et al. [48]	Architectural patterns	mu-calculus	Design and implementation	Automatic	Detecting SOA patterns
Wuyts [49]	Design patterns	Declarative reasoning	None	Manual	Finding structural relationships
Krämer et al. [10]	Design patterns	Prolog queries	None	Automatic	Architecture reconstruction
Tonella and Antoniol [50]	Design patterns	Concept analysis	None	Automatic	Pattern identification
Arévalo et al. [51]	Collaboration patterns	Concept analysis	None	Automatic	Detection of collaboration patterns
Guéhéneuc et al. [52]	Design patterns	Explanation-based CSP	Implementation	Automatic	Pattern identification
Alnusair et al. [53]	Design patterns	Semantic web technologies and first order logic	Design	Automatic	Pattern identification
Lucia et al. [54]	Design patterns	Model checking	None	Automatic	Pattern identification
Kaczor et al. [55]	Design patterns	Graph-based	None	Automatic	Pattern identification
von Detten [56]	Design patterns	Graph-based	None	Automatic	Pattern identification
Seeman and Gudenberg [57]	Design patterns	Graph-based	None	Automatic	Pattern identification

Table 5 – continued from previous page

Approach	Pattern types	Method	Pattern variants support	Automation	Focus
Balanyi and Ferenc [58]	Design patterns	Graph-based and DPML	None	Automatic	Pattern identification
Wendehals et al. [14]	Design patterns	Graph-rewrite and fuzzy logic	Design and implementation	Automatic	Pattern identification
Tsantalis et al. [59]	Design patterns	Similarity scoring between graph vertices	Implementation	Automatic	Pattern identification
Shull et al. [8]	Design patterns	Guidelines for manual identification	None	Manual	Architecture reconstruction
Bergenti and Poggi [7]	Design patterns	Interactive design assistance	None	Automatic	Design improvement
Palma et al. [60]	Design patterns	Goal-question-metric	None	Semiautomatic	Pattern recommendations
Guo et al. [41]	Design patterns	Rigi standard format	None	Semiautomatic	Architecture reconstruction
Heuzeroth et al. [9]	Design patterns	Custom detection algorithm per pattern	None	Automatic	Pattern identification
Washizaki et al. [61]	Design patterns	Comparing code before and after a pattern's introduction	None	Semiautomatic	Pattern identification
Pinzger and Gall [62]	Design patterns	String-pattern-matching	Implementation	Automatic	Architecture reconstruction
Rasool and Mäder [63]	Design patterns	SQL queries and RegEx	Implementation	Automatic	Pattern identification
Stencel and Wegrzynowicz [64]	Design patterns	First order logic translated to SQL queries	Design and Implementation	Automatic	Pattern identification
Keller et al. [65]	Design patterns	Model-based	Implementation	Automatic	Reverse engineering
Philippow et al. [11]	Design patterns	Minimal key structures	Implementation	Automatic	Pattern recovery
Our approach	Architectural patterns	DSL based	Design + implementation	Semiautomatic	Architecture evolution

Our approach can broadly be categorized as a software architecture reconstruction approach. Ducasse and Pollet [3] presented a survey on the state-of-the-art in the field of software architecture reconstruction. They analyze and categorize the existing approaches with respect to their goals, inputs, process, techniques, and outputs. They also discuss approaches that detect patterns during software architecture reconstruction.

8.1. Approaches based on architectural patterns

A number of other approaches have been presented that focus on the recovery of architectural pattern information. Medividovic et al. [42] combine architectural recovery

and architectural identification to create discovered and recovered architectural models and leverage architectural styles to identify and reconcile mismatches between them. In contrast to our approach, they do not consider the selection of the correct architectural styles. Our approach also does not require the systems requirements to be known and utilizes primitive information that is added by the architect to search for applicable architectural pattern instances.

DiscoTect [43], which is introduced by Yan et al. uses a state machine to automatically detect architectural styles in low level execution events during the runtime of a system. While this approach focuses on the behavioral information of a system, our semi-automatic approach currently focuses on the structure of a system while we plan to implement behavioral information as another source of knowledge about a system under maintenance in the future. Furthermore, the DiscoTect approach also faces the problem of false positives and detection rate.

Scaniello et al. [40] propose an approach for semi-automatically detecting layers in software systems based on the algorithm introduced by Kleinberg [66]. The authors implemented a prototype and provide a case study for JHotDraw [4]. While their approach is focused on semi-automatically detecting layers without prior knowledge, our approach is not limited to a specific pattern or pattern definition but also provides means to create and search for custom pattern definitions.

Sartipi describes a pattern-based approach for recovering software architecture [44]. It models the process as a graph pattern matching problem between an entity relationship graph and an architecture pattern graph. While this approach uses the two models as input, we use the source code and the architectural abstraction specification in the DSL as input and the resulting component models are only used for consistency checks.

Harris et al. [45] propose a style library and use a recognition engine to detect instances of these abstractions in the source code. While our approach also allows the definition of architectural patterns, we do not automatically recognize instances of this patterns but require the software architect to select the proposal that best fits his view of the software architecture. Furthermore, we introduce the intermediate step of our abstraction model that allows to define patterns on a higher abstraction level.

Lungu et al. [46] propose a visual architecture recovery approach that exploits the package structure of a software system. For this purpose, they introduce package patterns which they automatically detect based on heuristics. While our approach also exploits the package structure of a software system, it is not limited to package information and also supports other options for creating architecture abstractions.

Paakki et al. [47] present an approach for the detection of architectural and design patterns. It treats pattern detection as constraint satisfaction problem (CSP) and uses the AC-3 algorithm [67] to find pattern candidates and then use software metrics to assess the quality of the systems architecture. The approach is implemented in Java but uses a Prolog variant for the representation of architectures and patterns. Our approach is semiautomatic and our search algorithm has a worst-case time complexity of $O(n \times m)$, where n is the size of the pattern catalog and m is the number

[4] `http://www.jhotdraw.org/`

of components in the architectural description. Their approach on the other hand, is fully automatic and uses the AC-3 algorithm which as a worst-case time complexity of $O(ed^3)$, where e is the number of arcs and d is the size of the domain.

The approach presented by Di Penta et al. [48] automatically analyzes SOAP messages to detect architectural patterns in a SOA system. It is based on model checking verifying patterns on a model of the system, where patterns are described as mu-calculus logic formulae. While their approach uses execution traces collected by a monitoring system as input, our approach uses structural information as input and allows the description of patterns using a simple DSL that also allows to specify variation points.

In this subsection we presented a number of approaches focusing on recovering or detecting architectural patterns with a majority of approaches focusing on the automatic detection of patterns. As already discussed in Section 1 automatic approaches are limited by a high number of false positives while existing semiautomatic approaches focus either on a specific pattern [40] or are limited to specific languages [46]. In contrast, our approach focuses on architecture documentation and evolution of architectural patterns. It provides support for pattern variants and does not have the drawback of finding many false positives as it is semiautomatic and requires the software architect to annotate the architecture model with architectural primitive information.

8.2. Approaches based on design patterns

In this subsection we describe selected approaches for the identification of design patterns ranging from manual identification techniques to automatic detection approaches.

In the first part we discuss approaches that utilize formal methods like concept analysis [68, 51] or constraint satisfaction problems [52] to tackle design pattern identification. The second part contains approaches that represent patterns as graphs and or treat the problem of pattern identification as a graph matching problem [59]. In the last part we discuss approaches that use various other techniques for the identification patterns like String pattern matching [62] or SQL queries [63].

8.2.1. Approaches based on logic oriented programming / formal methods

In this section we present a number of approaches that use logic oriented programming or formal methods to identify design patterns in source code. Wuyts [49] uses declarative reasoning to find structural relationships in Smalltalk programs. He created a declarative framework for describing the structure of an object oriented system that he then uses to describe design patterns. However his approach is not directly focused on reconstructing patterns while our approach works on an architectural component view and finds architectural patterns using primitives.

Another approach is presented by Krämer et al. [10] that uses Prolog queries on C++ header files to find a number of structural design patterns. However the precision of their approach is only about 40 percent. Our approach is semi-automatic and based on primitive information that was annotated by the software architect. As we focus on architectural patterns, our approach works at a different level of abstraction.

Tonella and Antoniol [69] introduce an approach for object oriented pattern interference. It utilizes concept analysis to detect design patterns in C++ source code.

The authors present three case studies that showcase the approach. Their approach is automatic and works on a lower abstraction level than our semi-automatic approach focusing on architectural patterns.

Arévalo et al. [51] also use a formal approach based on concept analysis to detect collaboration patterns between software artifacts. Their approach is language independent and analyzes a system's structure and improve the pattern detection algorithm introduced by Tonella and Antoniol [69]. In comparison to our approach, their approach works on a lower abstraction level and focuses on class relations, while our approach is on a higher abstraction level and focuses on architectural patterns.

Guéhéneuc et al [52] propose a explanation-based constraint programming approach for identifying and correcting micro-architectures that are similar to design patterns. They use a library of constraints to search for design patterns. Their approach provides the benefit of being able to explain why pattern candidates where rejected. Based on these explanations their approach lets the user decide to relax specific constraints if desired and create new solutions. While our approach is at the moment not capable of giving the user feedback about rejected pattern candidates, our approach takes pattern variants into account and works on a higher abstraction level (components instead of classes) and thus a smaller search space.

Alnusair et al. [53] propose an approach that uses semantic web technologies for the detection of design patterns in source code. They formalize the structure and behavior of patterns using first order predicate logic. Our approach uses a model-driven representation of patterns and specifically focuses on the semiautomatic documentation and evolution of architectural patterns.

Lucia et al. [54] present an approach based on linear temporal logic that analyzes pattern instances' behavior statically and dynamically. First a set of pattern candidates is computed based on structural information. For the pattern candidates the model checking tool SPIN is used to check if they fulfill the behavior specified by in the pattern description using sequence diagrams. While their approach uses structural and behavioral information to automatically find design patterns, our approach uses structural information on a higher abstraction level to semiautomatically search for architectural pattern candidates.

All the approaches in this section utilize formal methods for the identification or the description of design patterns. With the exception of one [49], all these approaches are automatic and thus have a potentially slow run-time behavior and possibly yield a high number of false positives while our approach is semiautomatic and searches for architectural patterns on the level of architectural component which reduces the search space.

8.2.2. Graph-based approaches

This section presents different approaches that represent pattern descriptions or source code relations as graphs or use graph matching algorithms.

An approach that uses operations on finite sets of bit-vectors to detect design patterns is introduced by Kaczor et al. [55]. They utilizes the parallelism of bit-wise operations in a bit-vector algorithm that is able to detect exact as well as approximate instances of a pattern. The program and the patterns are represented as digraphs from

which a string representation is computed. These string representations are used as input for the bit-vector algorithm. They present 3 case studies where they search for the Composite and AbstractFactory patterns in three different existing applications. Our approach is model-based, works on a higher abstraction level, and uses an editable pattern catalog as input which is also used to check for pattern violations in an evolving software system.

von Detten [56] proposes to use symbolic execution in order to improve the detection of behavioral design patterns which are specified on the basis of UML sequence diagrams. The author integrates his approach into the Reclipse tool. Our approach, on the other hand, focuses on the documentation of architectural patterns which are detected based on structural information.

An approach for pattern-based design recovery in Java was introduced by Seemann and von Gudenberg [57]. Their approach is based on graphs and the authors showcase their approach for the detection of the design patterns Composite, Bridge, and Strategy. While their approach, like our approach, also detects patterns based on a system's structure, our approach works on a different level of abstraction and uses an editable, DSL-based pattern catalog.

Balanyi and Ferenc [58] introduce a design pattern description language DPML that is based on XML and allows the description of a patterns structure and behavior. They then use the C++ reverse engineering framework Columbus to create an abstract semantic graph of the software system and compare this graph to the pattern descriptions. Our approach works on a higher abstraction level, which is also reflected in the used pattern description languages. While our pattern descriptions are based on components, their DPML is based on classes and the interactions between classes.

Wendehals et al. [14] present an approach that they use to identify design patterns in Java source code. It uses graph rewrite rules and fuzzy logic to allow for design- and implementation variants of design patterns and utilizes dynamic analysis to improve the recognition of design patterns. The approach confirms the candidates that are found by static analysis using dynamic analysis. The confirmed pattern candidates are then presented to the user. Whereas this approach is based on graph rewrite rules and fuzzy logic to find design patterns in Java source code, our approach is semiautomatic and is based on DSLs that include support for variability.

Tsantalis et al. [59] propose a pattern detection approach based on the similarity scoring between graph vertices. Due to the underlying algorithm it supports the recognition of patterns that deviate from the defined structure. However in the presented version of their implementation, pattern descriptions are hard-coded within the tool, while our patterns are collected in a reusable pattern catalog that includes support for pattern variants.

All of the graph-based approaches that we presented in this section work automatically and focus on the identification of design patterns in source code. Our approach is semi-automatic and focuses on the documentation of architectural patterns as architectural knowledge based on architectural components. It specifically supports consistency checking of documented patterns in order to keep architectural documentation and source code consistent throughout the evolution of a software system.

8.2.3. Miscellaneous approaches

While a considerable number of approaches are based on formal methods or graphs, various approaches have been proposed that use different techniques for the identification of design patterns.

Shull et al. [8] proposed an inductive method that is aimed at aiding the manual identification of design patterns in a software design. They introduce a set of procedures and guidelines to aid the engineers who have no knowledge in the architecture reconstruction process.

Bergenti and Poggi [7] present an interactive design assistant that automatically finds a subset of the GoF patterns [12] in UML diagrams and produces critiques for the found patterns thus providing suggestions for design improvements. Our approach in contrast requires the manual annotation with primitive information and then automatically finds reusable architectural patterns in the annotated architectural component view. Thus our approach requires more effort from the software architect but does not have the issues encountered with automatic approaches.

A recommendation system for design patterns is proposed by Palma et al. [60]. They implement a design pattern recommender based on a goal-question-metric that focuses on supporting the engineers during the implementation / adaptation of a system. Based on the answers a user provides, the system suggests the pattern with the highest weight. While their approach is also semiautomatic, their systems needs the user to repeatedly answer it the questions in order to recompute the weights. Our approach, on the other hand, only requires the annotation of architectural primitives once.

Guo et al. [41] present an approach for architecture recovery and conformance checking called "ARM". This approach automatically searches for instances of patterns in a source model using query tools in Dali. Like all automatic approaches it requires manual correction of false positives and cannot guarantee a hundred percent detection rate, while our approach is semiautomatic and operates on a higher level of abstraction.

Heuzeroth et al. [9] detect design patterns in legacy code and classify their found pattern candidates based on the evidence they find during static and dynamic analyses. They present the application of their approach in two case studies. While their approach focuses on the detection of design patterns, our approach focuses on the documentation of architectural patterns and the consistency of the source code with the documented architectural information during a software system's evolution.

The pattern detection approach introduced by Washizaki et al. [61] compare the source code of a system before and after an application of a design pattern. They detect structural design patterns and are able to distinguish between structurally equal patterns based on systems behavior. However their approach needs the user specify if the system matches the conditions and smells of a pattern manually. Our approach tests all patterns in our pattern catalog and suggests pattern candidates automatically after the user has annotated the architectural component view with architectural primitive information.

Pinzger and Gall [62] use an interactive and iterative architecture recovery approach that is built upon low level patterns and creates pattern views. The pattern views are then used to abstract higher level patterns which enable the description of a system's architecture. In contrast to this approach where design patterns are automatically de-

tected in the source code via string-pattern-matching, our approach allows to semiautomatically define abstractions, and then provides the architect with a number of possible existing patterns.

An automated pattern identification approach that uses reusable feature types to support pattern variants is presented by Rasool and Mäder [63]. This approach uses regular expressions and SQL queries on source models while our approach is semiautomatic and operates on architectural primitives and architectural abstraction specifications.

A pattern identification approach that uses SQL statements to identify selected patterns and claims support for pattern variants is presented by Stencel and Wegrzynowicz [64]. However this approach is automatic and does not allow to explicitly specify variation points (except those that SQL already allows) but only allows for implicit variants, while our approach is semiautomatic and additionally allows explicit variants.

An approach that supports automatic as well as semiautomatic and manual identification of design patterns is presented by Keller et al. [65]. They provide a case study with three real world systems where they identified the instances of three exemplary design patterns. In constrast, our approach works on a higher level of abstraction and allows the documentation of architectural patterns after they have been found. Furthermore our approach supports variability in a patterns description, while their approach requires the user to modify the design queries that identify patterns.

Philippow et al. [11] give an overview over existing automatic design pattern recovery approaches and introduce an approach based on minimal key structures that uses a number of positive and negative search criteria including the possibility of uncertain elements. They implemented the search algorithms for the GoF patterns [12] Composite, Singleton, and Interpreter. While their approach and our approach share similarities with regard to variability support, their approach uses fixed algorithms for design patterns in contrast to our approach that allows the customization and definition of architectural patterns.

9. Conclusion

In this article we have presented an approach for the semi-automatic documentation of architectural patterns based on architectural primitives. While other approaches automatically detect design patterns in the source code, we require the architect to semi-automatically create an abstraction of a architectural component view that is annotated with architecture primitive information. This raises the abstraction level of the input on which we automatically search for patterns. It also reduces the number of found pattern candidates as well as the search space for our automatic *Pattern Instance Documentation Tool*, as the number of architectural components is significantly smaller then the number of objects in a system. As we use architectural primitives as the basis for our pattern templates in our pattern catalog and in the architectural component view, our search can make use of this additional architectural information and the constraints that are captured by these primitives. Once a pattern instance is documented, our approach subsequently performs automatic consistency checking. We applied our approach in three open source systems case studies to show the applicability of the approach. Our pattern catalog supports the definition of patterns based on primitives, is

reusable, supports pattern variability, and can be customized and is extensible. To use our approach, an initial investment in creating a pattern catalog (patterns and pattern variants) is required. Our performance evaluation results show that the approach is applicable on a typical developer machine during software design and development, even for very large model sizes.

Our case studies (Section 5) show that the concept of architectural primitives can be applied to document architectural patterns during architecture reconstruction (RQ1) and support the further architecture evolution once they are documented (RQ2, as exemplified in the Frag case study in Section 5.2).

As we could apply our approach for three systems which all implement different types of applications with different project sizes, it is likely that our results can be generalized to other similar cases (RQ3). We could show the feasibility of the approach with the implementation of our prototype which was used to perform the case studies described in this paper (RQ1, RQ2) and is described in detail in Section 4. In addition to this, the performance evaluation of our prototype (see Section 6) shows that algorithms for semi-automatic identification of architectural patterns are sufficiently efficient to be used for architectural pattern identification on common developer computers and thus should be sufficiently efficient for day-to-day use (RQ4). With respect to RQ5 we found only a small effort was necessary for evolving our pattern catalog during the case studies.

We plan to further investigate approaches for collaboratively editing and sharing the pattern catalog among users and in the community. At the moment we only support structural primitives that either annotate components or their connectors but no behavioral information. The additional integration of behavior primitives, as for instance introduced by Kamal et al. [70], is a topic for future research.

References

[1] A. Jansen, J. van der Ven, P. Avgeriou, D. K. Hammer, Tool Support for Architectural Decisions, in: Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture, WICSA '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 4–14. doi:10.1109/WICSA.2007.47.

[2] D. Rost, M. Naab, C. Lima, C. von Flach Garcia Chavez, Software architecture documentation for developers: A survey, in: K. Drira (Ed.), Software Architecture, Vol. 7957 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 72–88. doi:10.1007/978-3-642-39031-9_7.
URL http://dx.doi.org/10.1007/978-3-642-39031-9_7

[3] S. Ducasse, D. Pollet, Software Architecture Reconstruction: A Process-Oriented Taxonomy, Software Engineering, IEEE Transactions on 35 (4) (2009) 573–591.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture: A System of Patterns, John Wiley & Sons, Inc., New York, NY, USA, 1996.

[5] K. Beck, R. E. Johnson, Patterns Generate Architectures, in: Proceedings of the 8th European Conference on Object-Oriented Programming, ECOOP '94, Springer-Verlag, London, UK, UK, 1994, pp. 139–149.

[6] D. B. Lange, Y. Nakamura, Interactive Visualization of Design Patterns Can Help in Framework Understanding, in: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications, OOPSLA '95, ACM, New York, NY, USA, 1995, pp. 342–357. doi:10.1145/217838.217874.

[7] F. Bergenti, A. Poggi, Improving UML Designs Using Automatic Design Pattern Detection, in: In Proc. 12th. International Conference on Software Engineering and Knowledge Engineering (SEKE 2000), 2000, pp. 336–343.

[8] F. Shull, W. Melo, V. Basili, An Inductive Method for Discovering Design Patterns from Object-oriented Software Systems, Computer science technical report series, University of Maryland, 1996.

[9] D. Heuzeroth, T. Holl, G. Högström, W. Löwe, Automatic Design Pattern Detection, in: Proceedings of the 11th IEEE International Workshop on Program Comprehension, IWPC '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 94–104.

[10] C. Krämer, L. Prechelt, Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software, in: Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96), WCRE '96, IEEE Computer Society, Washington, DC, USA, 1996, pp. 208–216.

[11] I. Philippow, D. Streitferdt, M. Riebisch, Design Pattern Recovery in Architectures for Supporting Product Line Development and Application, in: Modelling Variability for Object-Oriented Product Lines, BookOnDemand Publ. Co, 2003, pp. 42–57.

[12] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[13] L. Wendehals, J. Niere, J. P. Wadsack, Design Pattern Recovery Based on Source Code Analysis with Fuzzy Logic, Tech. rep., University of Paderborn (2001).

[14] L. Wendehals, Improving Design Pattern Instance Recognition by Dynamic Analysis, in: Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA, 2003, pp. 29–32.

[15] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, R. Little, Documenting Software Architectures: Views and Beyond, Pearson Education, 2002.

[16] U. Zdun, P. Avgeriou, Modeling Architectural Patterns Using Architectural Primitives, in: Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 2005), ACM Press, San Diego, CA, USA, 2005, pp. 133–146.

[17] R. N. Taylor, N. Medvidovic, E. M. Dashofy, Software Architecture - Foundations, Theory, and Practice, Wiley, 2010.

[18] P. Avgeriou, U. Zdun, Architectural Patterns Revisited - A Pattern Language, in: Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), Irsee, Germany, 2005, pp. 1–39.

[19] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Addison-Wesley, 1996.

[20] N. Medvidovic, R. N. Taylor, A classification and comparison framework for software architecture description languages, IEEE Trans. Softw. Eng. 26 (2000) 70–93. doi:10.1109/32.825767.

[21] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, J. E. Robbins, Modeling software architectures in the Unified Modeling Language, ACM Trans. Softw. Eng. Methodol. 11 (1) (2002) 2–57.

[22] A. H. Eden, Y. Hirshfeld, LePUS – Symbolic Logic Modeling of Object Oriented Architectures: A Case Study, in: Second Nordic Workshop on Software Architecture - NOSA'99, Ronneby, Sweden, 1999, pp. 1–14.

[23] T. Mikkonen, Formalizing Design Patterns, in: Proceedings of the 20th international conference on Software engineering, IEEE Computer Society, Kyoto, Japan, 1998, pp. 115–124.

[24] N. R. Mehta, N. Medvidovic, Composing Architectural Styles From Architectural Primitives, in: Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press, Helsinki, Finland, 2003, pp. 347–350.

[25] T. Haitzer, U. Zdun, DSL-based Support for Semi-Automated Architectural Component Model Abstraction Throughout the Software Lifecycle, in: Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures, QoSA '12, ACM, New York, NY, USA, 2012, pp. 61–70.

[26] The Freecol Team, FreeCol, http://freecol.org (2011).

[27] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[28] Apache CXF, http://cxf.apache.org (2011).

[29] U. Zdun, M. Kircher, M. Völter, Remoting Patterns, IEEE Internet Computing 8 (6) (2004) 60–68.

[30] U. Zdun, The Frag Language, http://frag.sourceforge.net/ (2011).

[31] E. Freeman, E. Freeman, B. Bates, K. Sierra, Head First Design Patterns, O' Reilly & Associates, Inc., 2004.

[32] E. Curry, D. Chambers, G. Lyons, Extending message-oriented middleware using interception, in: 3rd International Workshop on Distributed Event-Based Systems (DEBS'04), Edinburgh, Scotland, UK, 2004, pp. 32–37.

[33] D. C. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, 2nd Edition, John Wiley & Sons, Inc., New York, NY, USA, 2000.

[34] C. Xu, Y. Liu, S. Cheung, C. Cao, J. Lv, Towards context consistency by concurrent checking for internetware applications, Science China Information Sciences 56 (8) (2013) 1–20. doi:10.1007/s11432-013-4907-5.

[35] C. L. Heitmeyer, R. D. Jeffords, B. G. Labaw, Automated consistency checking of requirements specifications, ACM Trans. Softw. Eng. Methodol. 5 (3) (1996) 231–261. doi:10.1145/234426.234431.

[36] R. Mahajan, B. Shneiderman, Visual and textual consistency checking tools for graphical user interfaces, IEEE Trans. Softw. Eng. 23 (11) (1997) 722–735. doi:10.1109/32.637386.

[37] M. A. Javed, U. Zdun, The supportive effect of traceability links in architecture-level software understanding: Two controlled experiments, in: 2014 IEEE/IFIP Conference on Software Architecture, WICSA 2014, Sydney, Australia, April 7-11, 2014, 2014, pp. 215–224. doi:10.1109/WICSA.2014.43.

[38] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, Experimentation in Software Engineering, Springer, 2012. doi:10.1007/978-3-642-29044-2.

[39] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, J. Rosenberg, Preliminary Guidelines for Empirical Research in Software Engineering, IEEE Trans. Softw. Eng. 28 (8) (2002) 721–734.

[40] G. Scanniello, A. D'Amico, C. D'Amico, T. D'Amico, An Approach for Architectural Layer Recovery, in: Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10, ACM, New York, NY, USA, 2010, pp. 2198–2202. doi:10.1145/1774088.1774551.

[41] G. Y. Guo, J. M. Atlee, R. Kazman, A Software Architecture Reconstruction Method, in: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1), Kluwer, B.V., Deventer, The Netherlands, The Netherlands, 1999, pp. 15–34.

[42] N. Medvidovic, A. Egyed, P. Grünbacher, Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery, in: Proc. of the 2nd International Software Requirements to Architectures Workshop(STRAW), Portland, Oregon, 2003, pp. 61–68.

[43] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, R. Kazman, DiscoTect: A System for Discovering Architectures from Running Systems, in: Proceedings of the 26th International Conference on Software Engineering, ICSE '04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 470–479.

[44] K. Sartipi, Software Architecture Recovery based on Pattern Matching, in: Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on, Washington, DC, USA, 2003, pp. 293–296.

[45] D. R. Harris, H. B. Reubenstein, A. S. Yeh, Reverse Engineering to the Architectural Level, in: Proceedings of the 17th international conference on Software engineering, ICSE '95, ACM, New York, NY, USA, 1995, pp. 186–195.

[46] M. Lungu, M. Lanza, T. Girba, Package Patterns for Visual Architecture Recovery, in: Proceedings of the Conference on Software Maintenance and Reengineering, CSMR '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 185–196.

[47] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, A. I. Verkamo, Software Metrics by Architectural Pattern Mining, in: in Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress, 2000, pp. 325–332.

[48] M. Di Penta, A. Santone, M. L. Villani, Discovery of SOA Patterns via Model Checking, in: 2nd International Workshop on Service Oriented Software Engineering: In Conjunction with the 6th ESEC/FSE Joint Meeting, IW-SOSWE '07, ACM, New York, NY, USA, 2007, pp. 8–14.

[49] R. Wuyts, Declarative Reasoning about the Structure of Object-Oriented Systems, in: Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings, 1998, pp. 112–124. doi:10.1109/TOOLS.1998.711007.

[50] P. Tonella, G. Antoniol, Inference of object-oriented design patterns, Journal of Software Maintenance 13 (5) (2001) 309–330.

[51] G. Arevalo, F. Buchli, O. Nierstrasz, Detecting Implicit Collaboration Patterns, in: Reverse Engineering, 2004. Proceedings. 11th Working Conference on, 2004, pp. 122–131. doi:10.1109/WCRE.2004.18.

[52] Y.-G. Guéhéneuc, N. Jussien, Using explanations for design-patterns identification, in: proceedings of the 1 st IJCAI workshop on Modeling and Solving Problems with Constraints, AAAI Press, 2001, pp. 57–64.

[53] A. Alnusair, T. Zhao, G. Yan, Automatic recognition of design motifs using semantic conditions, in: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, ACM, New York, NY, USA, 2013, pp. 1062–1067. doi:10.1145/2480362.2480564.

[54] A. De Lucia, V. Deufemia, C. Gravino, M. Risi, Improving Behavioral Design Pattern Detection through Model Checking, in: Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on, 2010, pp. 176–185. doi:10.1109/CSMR.2010.16.

[55] O. Kaczor, Y. Guéhéneuc, S. Hamel, Efficient Identification of Design Patterns with Bit-vector Algorithm, in: Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on, 2006, pp. 10 pp.–184. doi:10.1109/CSMR.2006.25.

[56] M. von Detten, Towards Systematic, Comprehensive Trace Generation for Behavioral Pattern Detection Through Symbolic Execution, in: Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools, PASTE '11, ACM, New York, NY, USA, 2011, pp. 17–20. doi:10.1145/2024569.2024573.

[57] J. Seemann, J. W. von Gudenberg, Pattern-based Design Recovery of Java Software, in: Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '98/FSE-6, ACM, New York, NY, USA, 1998, pp. 10–16.

[58] Z. Balanyi, R. Ferenc, Mining Design Patterns from C++ Source Code, in: Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on, 2003, pp. 305–314. doi:10.1109/ICSM.2003.1235436.

[59] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. Halkidis, Design Pattern Detection Using Similarity Scoring, Software Engineering, IEEE Transactions on 32 (11) (2006) 896–909. doi:10.1109/TSE.2006.112.

[60] F. Palma, H. Farzin, Y. Gueheneuc, N. Moha, Recommendation System for Design Patterns in Software Development: An DPR Overview, in: Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on, 2012, pp. 1–5. doi:10.1109/RSSE.2012.6233399.

[61] H. Washizaki, K. Fukaya, A. Kubo, Y. Fukazawa, Detecting design patterns using source code of before applying design patterns, in: Proceedings of the 2009 Eigth IEEE/ACIS International Conference on Computer and Information Science, ICIS '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 933–938. doi:10.1109/ICIS.2009.209.

[62] M. Pinzger, H. Gall, Pattern-Supported Architecture Recovery, in: Proceedings of the 10th International Workshop on Program Comprehension, IWPC '02, IEEE Computer Society, Washington, DC, USA, 2002, pp. 53–62.

[63] G. Rasool, P. Mader, Flexible Design Pattern Detection Based on Feature Types, in: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 243–252. doi:10.1109/ASE.2011.6100060.

[64] K. Stencel, P. Wegrzynowicz, Detection of Diverse Design Pattern Variants, in: Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific, 2008, pp. 25–32. doi:10.1109/APSEC.2008.67.

[65] R. K. Keller, R. Schauer, S. Robitaille, P. Pagé, Pattern-Based Reverse-Engineering of Design Components, in: Proceedings of the 21st international conference on Software engineering, ICSE '99, ACM, New York, NY, USA, 1999, pp. 226–235. doi:10.1145/302405.302622.

[66] J. M. Kleinberg, Authoritative Sources in a Hyperlinked Environment, J. ACM 46 (1999) 604–632. doi:10.1145/324133.324140.

[67] A. K. Mackworth, Consistency in Networks of Relations, Artificial Intelligence 8 (1) (1977) 99–118. doi:http://dx.doi.org/10.1016/0004-3702(77)90007-8.

[68] P. Tonella, G. Antoniol, Object Oriented Design Pattern Inference, in: Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on, 1999, pp. 230–238. doi:10.1109/ICSM.1999.792619.

[69] P. Tonella, G. Antoniol, Object Oriented Design Pattern Inference, Journal of Software Maintenance and Evolution: Research and Practice 13 (5) (2001) 309–330.

[70] A. W. Kamal, P. Avgeriou, Modeling Architectural Patterns' Behavior Using Architectural Primitives, in: Proceedings of the 2nd European conference on Software Architecture, ECSA '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 164–179. doi:10.1007/978-3-540-88030-1_13.

Appendix A. Xtext grammar of the Architecture Abstraction DSL

```
grammar at.ac.univie.cs.swa.component.architectureabstraction.
    ArchitectureAbstractionDSL
with org.eclipse.xtext.common.Terminals

generate architectureAbstractionDSL
"http://www.univie.ac.at/cs/swa/component/architectureabstraction/
    ArchitectureAbstractionDSL"
import "http://www.eclipse.org/uml2/4.0.0/UML" as umlMM
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
import "http://www.ac.at/univie/cs/swa/pattern/catalog/
    PatternCatalogDSL" as patcat

Transformation:
  name=STRING
  components+=(ComponentDef)+;

QUALIFIED_NAME returns ecore::EString:
  ID ("." ID)*;

ComponentDef returns ComponentDef:
```

```
  'Component' name=ID
  'consists of'
  (expr=OrComposition)
  annotations+=ComponentAnnotation*
  connectors+=ComponentConnector*;

OrComposition returns Expression:
  ExcludeComposition ({OrComposition.left=current} 'or' right=
      ExcludeComposition)*;

ExcludeComposition returns Expression:
  AndComposition ({ExcludeComposition.left=current} 'and not' right=
      Primary)*;

AndComposition returns Expression:
  Primary ({AndComposition.left=current} 'and' right=Primary)*;

Primary returns Expression:
  NameFilter |
  RelationFilter |
  ExtensionFilter |
  '{' OrComposition '}';

NameFilter:
  PackageNameFilter | ClassNameFilter;

RelationFilter:
  ContainedInPackage | UsesFilter | UsedByFilter | ChildOfFilter |
      Supertype | InstanceOf | IsClass | SpecificInterface;

PackageNameFilter:
  'Package' '(' regEx=STRING ')';

ClassNameFilter:
  'Class' '(' regEx=STRING ')';

UsesFilter:
  'Uses' '(' relatedTo=[umlMM::Classifier|QUALIFIED_NAME] ')';

UsedByFilter:
  'UsedBy' '(' relatedTo=[umlMM::Classifier|QUALIFIED_NAME] ')';

ChildOfFilter:
  'ChildOf' '(' relatedTo=[umlMM::Class|QUALIFIED_NAME] ')';

Supertype:
  'Supertype' '(' relatedTo=[umlMM::Class|QUALIFIED_NAME] ')';

ContainedInPackage:
  'Package' '(' relatedTo=[umlMM::Package|QUALIFIED_NAME] (','
      excludeChildren?='excludeChildren')? & (',' excludeNestedElements
      ?='excludeNestedElements')?) ')';

IsClass:
  'Class' '(' relatedTo=[umlMM::Class|QUALIFIED_NAME] (','
      excludeChildren?='excludeChildren')?  ')';
```

```
InstanceOf:
  'InstanceOf' '(' relatedTo=[umlMM::Interface|QUALIFIED_NAME] (',' 
      excludeInterface?='excludeInterface')? ')';

SpecificInterface:
  'Interface' '(' relatedTo=[umlMM::Interface|QUALIFIED_NAME] (',' 
      excludeChildren?='excludeChildren')?  ')';

ExtensionFilter:
  JavaExtensionFilter | XtendExtensionFilter;

JavaExtensionFilter:
  'Java' '(' staticMethod=STRING ')';

XtendExtensionFilter:
  'Xtend' '(' function=STRING ')';

/* primitive annotations */
PrimitiveAnnotation:
  name=ID;

MyPrimitiveAnnotation returns PrimitiveAnnotation:
  ConnectorAnnotation |
  ComponentAnnotation;

ComponentAnnotation returns PrimitiveAnnotation:
  GroupingAnnotation | LayeringAnnotation;

ConnectorAnnotation:
  AggregationCascadeAnnotation | CompositeCascadeAnnotation |
      CallbackAnnotation | IndirectionAnnotation |
      VirtualConnectorAnnotation | SimpleConnectorAnnotation |
      ShieldAnnotation | TypingAnnotation;

ComponentConnector:
  {ComponentConnector}
  annotation=ConnectorAnnotation
  ('connector name: ' connectorName=ID)?
  ('implemented by'
  //  implementationExpression=[umlMM::Relationship|QUALIFIED_NAME]
  (implementingExpression+=OrComposition)?
  ('relation: ' implementingRelations+=[umlMM::Dependency|
      QUALIFIED_NAME]
  (',' implementingRelations+=[umlMM::Dependency|QUALIFIED_NAME])*)?)?;

GroupingAnnotation returns PrimitiveAnnotation:
  {GroupingAnnotation} 'is in Group (' groupId=ID ')';

LayeringAnnotation returns GroupingAnnotation:
  {LayeringAnnotation} 'is at Layer (' groupId=ID ')';

ShieldAnnotation returns ConnectorAnnotation:
  {ShieldAnnotation}
  "is a Shield for" (targets+=[PatternInstancePrimitiveTarget] (',' 
      targets+=[PatternInstancePrimitiveTarget])*)?;

TypingAnnotation returns ConnectorAnnotation:
```

```
  {TypingAnnotation}
  "is Typing for" (targets+=[PatternInstancePrimitiveTarget] (','
      targets+=[PatternInstancePrimitiveTarget])*)?;

SimpleConnectorAnnotation returns ConnectorAnnotation:
  {SimpleConnectorAnnotation}
  'connector to' (targets+=[PatternInstancePrimitiveTarget] (','
      targets+=[PatternInstancePrimitiveTarget])*)?;

VirtualConnectorAnnotation returns ConnectorAnnotation:
  {VirtualConnectorAnnotation}
  'virtually connected to' (targets+=[PatternInstancePrimitiveTarget] (
      ',' targets+=[PatternInstancePrimitiveTarget])*)?;

IndirectionAnnotation returns ConnectorAnnotation:
  {IndirectionAnnotation}
  'indirection to' (targets+=[PatternInstancePrimitiveTarget] (','
      targets+=[PatternInstancePrimitiveTarget])*)?;

CallbackAnnotation returns ConnectorAnnotation:
  {CallbackAnnotation}
  'callback with' target=[PatternInstancePrimitiveTarget] 'trigger
      interface' triggerInterface=[umlMM::Interface|QUALIFIED_NAME] '
      callback interface' callbackInterface=[umlMM::Interface|
      QUALIFIED_NAME];

CompositeCascadeAnnotation returns AggregationCascadeAnnotation:
  {CompositeCascadeAnnotation}
  'composition of:' targets+=[ComponentDef] (','targets+=[ComponentDef
      ])*;

AggregationCascadeAnnotation returns IndirectionAnnotation:
  {AggregationCascadeAnnotation}
  'aggregation of:'  targets+=[ComponentDef] (','targets+=[ComponentDef
      ])*;

PatternInstancePrimitiveTarget: ComponentDef | GroupingAnnotation;
```

Listing 1: Excerpt of the Xtext grammar of our architectural abstraction DSL