Science of Computer Programming 90 (2014) 135–160

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Semi-automated architectural abstraction specifications for supporting software evolution

Thomas Haitzer[*], Uwe Zdun

Software Architecture Group, Faculty of Computer Science, University of Vienna, Vienna, Austria

HIGHLIGHTS

- We provide semi-automatic architecture abstractions based on UML-component models.
- Stable architecture abstractions with respect to software evolution.
- Built-in support for traceability and consistency checking.

ARTICLE INFO

Article history:
Received 15 October 2012
Received in revised form 11 October 2013
Accepted 12 October 2013
Available online 24 October 2013

Keywords:
Architectural abstraction
Architectural component and connector views
Software evolution
UML
Model transformation

ABSTRACT

In this paper we present an approach for supporting the semi-automated architectural abstraction of architectural models throughout the software life-cycle. It addresses the problem that the design and implementation of a software system often drift apart as software systems evolve, leading to architectural knowledge evaporation. Our approach provides concepts and tool support for the semi-automatic abstraction of architecture component and connector views from implemented systems and keeping the abstracted architecture models up-to-date during software evolution. In particular, we propose architecture abstraction concepts that are supported through a domain-specific language (DSL). Our main focus is on providing architectural abstraction specifications in the DSL that only need to be changed, if the architecture changes, but can tolerate non-architectural changes in the underlying source code. Once the software architect has defined an architectural abstraction in the DSL, we can automatically generate architectural component views from the source code using model-driven development (MDD) techniques and check whether architectural design constraints are fulfilled by these models. Our approach supports the automatic generation of traceability links between source code elements and architectural abstractions using MDD techniques to enable software architects to easily link between components and the source code elements that realize them. It enables software architects to compare different versions of the generated architectural component view with each other. We evaluate our research results by studying the evolution of architectural abstractions in different consecutive versions of five open source systems and by analyzing the performance of our approach in these cases.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

In many software projects the design and the implementation drift apart during development and system evolution [1]. In some small projects this problem can be avoided, as it might be possible to understand and maintain a well written

[*] Corresponding author. Tel.: +43 1 4277 78521; fax: +43 1 4277 8 78521.
E-mail addresses: thomas.haitzer@univie.ac.at (T. Haitzer), uwe.zdun@univie.ac.at (U. Zdun).
URLs: http://informatik.univie.ac.at/thomas.haitzer (T. Haitzer), http://informatik.univie.ac.at/uwe.zdun (U. Zdun).

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2013.10.004

source code without additional architectural documentation. For many larger systems, this is not an option, and additional architectural documentation is required to aid the understanding of the system and especially to comprehend the "big picture" by providing architectural knowledge about a system's design [2]. One way to provide this information are automatically generated diagrams of the systems (e.g. in form of class diagrams) [3]. However these diagrams usually do not represent higher-level abstractions, and hence they hardly support the understanding of the big picture. First of all, the sheer size of the automatically generated diagrams is often a problem. In addition, creating an automatic layout or partitioning that is understandable is still an open research topic [4,5]. Clustering approaches from the reengineering research literature (e.g. [6–8]) can help to obtain an initial understanding and make sense of such diagrams. However the case study by Corazza et al. [9] shows that in five out of seven cases it is necessary to make manual corrections for about half of the entities of the analyzed source code.

As a consequence, today the documentation of the system's architecture is usually maintained manually. To model architectural knowledge, often models using box-and-line-diagrams [10], UML [11], architecture description languages (ADLs) [12], or similar modeling approaches are used. In many cases, such models are created before the actual implementation begins. Later, during implementation and system evolution, they loose touch with reality because changes to the software design are only made in the source code while the architectural models are not updated [13]. This problem is known as *architectural knowledge evaporation* [1].

Our approach focuses on architectural abstractions from the source code in a changing environment while still supporting traceability. It was initially introduced in a paper at the QoSA 2012 conference [14]. In this article we further extend our approach with respect to its traceability and consistency checking capabilities. We describe in detail how the approach provides these features to the software architect. Furthermore we provide extended case studies of our approach which also include additional information regarding traceability and consistency.

A considerable number of works exist that focus on abstractions from source code [15,16,8,17]. However, to the best of our knowledge, so far none of these approaches targets architectural abstractions at different levels of granularity, traceability between architectural models and the code, and the ability to cope with the constant evolution of software systems. Our approach introduces the semi-automatic abstraction of architectural component and connector view models from the source code based on an architectural abstraction specified in a domain specific language (DSL) [18,19]. In contrast to the related works, our approach specifically targets architectural abstractions and requires changes to the architectural abstraction specifications only in the rare case that the architecture of the system changes, but not for the vast majority of non-architectural changes we see during a software system's evolution (see Section 2). Please note that in the literature the term "component model" is often used to describe meta-models for component-based development [20]. In this paper, we (only) use the term *architectural* component and connector view (or component view for short) to describe a model that contains architectural components (as in [21]).

We chose a semi-automatic approach to enable the software architect to provide information which system details are relevant for getting the right level of abstraction – as software architecture is usually described in different views at different levels of abstraction. Our goal is to let the software architect specify this information with minimal effort in an easy-to-comprehend DSL that provides good tool support. Our approach allows architects to create different architectural abstraction specifications that represent different levels of abstraction and thus supports views ranging from high-level software architectural views to more low-level software design views. Once the software architect has defined an architectural abstraction in the DSL, we can automatically generate architectural component views from the source code using model-driven development (MDD) techniques and check whether architectural design constraints are fulfilled by these models.

As our approach focuses on defining stable abstractions in the architectural abstraction specification, it can cope with many changes to the underlying source code without changing the architecture description (i.e., an instance of the DSL). Only changes to the architecture itself, which usually require a substantial modification of the source code, require the architectural abstraction specification to be updated. By creating different versions of the architectural component view over time, we are able to use a delta comparison to check and reason about the changes of the architectural component view. The generated models can be compared to a design model, to check the consistency of an implementation and its design, and to analyze the differences. To support the iterative nature of our approach, it also supports automatically checking the consistency between the source code model and the architecture abstraction specification on the fly.

Once the architectural component views have been abstracted, another problem is to identify which parts of the source code contribute to a specific component, i.e., to support traceability between architectural models and code. Today, this usually requires substantial and non-trivial manual effort to identify which code elements are related to which model elements. In contrast, in our approach, traceability can be automatically ensured, as model-driven development (MDD) [22] is used to generate the required traceability links between the model elements and the source code directly from the architectural abstraction specification.

The remainder of the paper is organized as follows: Section 2 explains the research problem addressed by this paper in more detail, as well as the research method that was applied to design and evaluate the DSL. Section 3 gives an overview of our approach. Section 4 provides details about our architectural abstraction DSL and its implementation. In Section 5 we present the evaluation of our approach based on five cases and a performance evaluation. In Section 6 we discuss open issues and lessons learned. Section 7 compares our approach to the related work, and we conclude in Section 8.

2. Research problem and research method

During the software development life-cycle, the source code and the architecture of a system evolve and change. This often results in architectural knowledge evaporation [1]. One of the reasons for this is that in today's software development processes the software architects often have to manually capture and maintain the architectural knowledge, which is a tedious task that is often forgotten in the daily business [13]. Additionally, when using conventional means for architecture documentation like abstracted UML models or box-and-line diagrams, the traceability between the architecture and the source code is lost. This can also lead to architectural knowledge evaporation, when architects and developers lose track of the correspondences between code and architecture.

A number of approaches have been proposed to address this research problem by providing automatically or semi-automatically produced abstractions from the source code [15,16,8,17]. In contrast to these related works, our approach *specifically targets architectural abstractions*. That is, we have designed our DSL to only require changes of the architecture abstraction specification once the architecture of the underlying software system has changed, but not for other kinds of changes. In case of non-architectural changes, an updated architectural documentation can automatically be re-generated from the altered source code without manual changes in the architectural abstraction specification.

To reach this goal we have designed and implemented the DSL using an incremental refinement process, following the design science research method [23]. In design science research, first a research question is posed, and then the develop/evaluate cycle is continuously repeated until a satisfactory solution for the research question has been obtained. In the course of this research, the research question can be altered or refined. In the first iterations, usually simplifying assumptions are made, which are stepwise removed during later iterations.

For this approach, we formulated the following research questions:

RQ1 Is it possible to create architectural abstractions based on generic filters that are stable and that only have to be changed when architectural changes occur but do not have to be changed when non-architectural changes occur in order prevent knowledge evaporation and providing up-to-date architecture documentation throughout the evolution of a software system?

RQ2 Is it possible to automatically generate traceability links that can be used to support the architects in the definition of the architectural abstractions?

RQ3 Is it possible to automatically check the consistency between design and code for different versions to support the architect during evolution of a system?

RQ4 What is the effort to create and maintain architectural abstractions under the assumption that knowledge about a system's architecture already exists?

RQ5 Is it possible to do consistency checking and generate an architectural component view with traceability links in an acceptable amount of time on a development machine?

In our design science research, these research questions emerged incrementally. We started with RQ1 and incrementally refined it through the other research questions. In particular, we learned during our research that traceability links (RQ2) are important for creating architectural abstractions. Our focus on evolution in RQ1 later led us to also study consistency checking problems during system evolution (RQ3). Finally, our early prototype and use experiences showed that development effort (RQ4) and execution time (RQ5) are important for acceptance and usability of our approach.

To address the research questions, while developing our DSL, we have studied the evolution of various software systems and their architecture documentations. We have classified changes in these systems into architectural changes and non-architectural changes. In an incremental refinement process, we have improved the DSL and its DSL tools to only require changes to the architectural abstraction code for changes classified as architectural abstractions in the studied samples of architectural evolution. In each incremental design cycle we have added more samples of architectural evolution and continued the iterations until only architectural changes required changes in the architecture abstraction specification.

Finally, we have evaluated the resulting DSL for all changes that can be observed in a number of consecutive versions of five open source projects. As can be seen in this study, reported in Section 5, the vast majority of changes are non-architectural changes, and they can be tolerated by the architectural abstractions defined in our DSL without changes to the architectural abstractions. Only when changes that are classified as architectural changes are introduced in the open source systems, updates to the architectural abstraction code in the DSL are necessary.

3. Approach overview

In this section we present an overview of our approach for supporting the semi-automated architectural abstraction of architectural component views throughout the software life-cycle. Our approach allows software architects to compare the abstract model with a previously defined architectural model and to maintain that model in correspondence with the source code over time. For this purpose we have introduced a DSL that defines architectural abstractions from class models, which can be automatically extracted from the source code, into architectural component views. We believe that in a lot of cases it should be possible to create architectural abstractions that are stable during the implementation process and only needs to be changed when architectural changes occur (e.g., leading to significant restructuring of the architectural design).

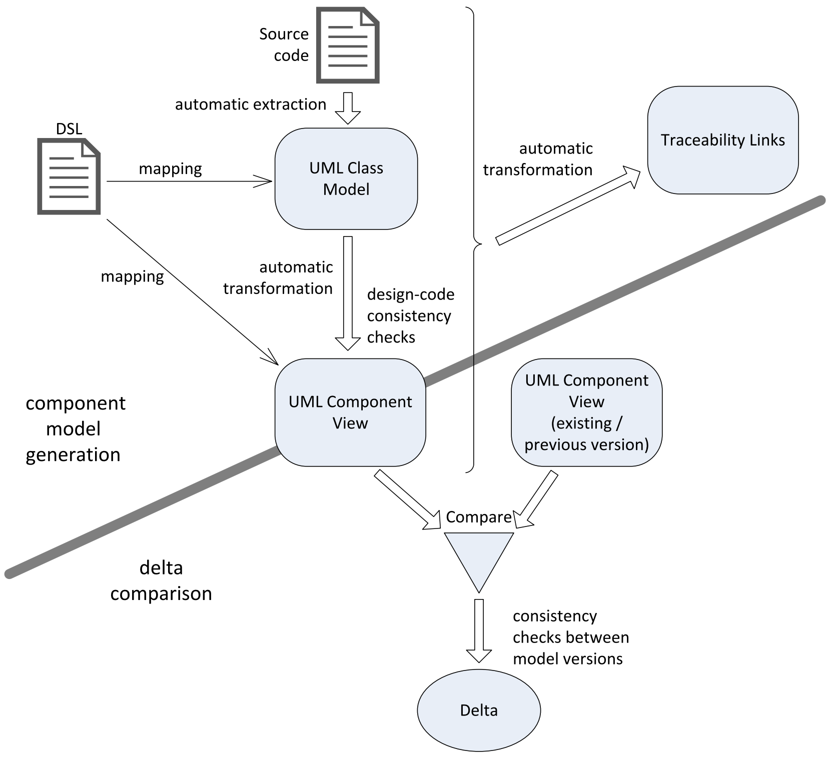


Fig. 1. Generating architectural component views from source code and comparing different model versions.

Once an architectural abstraction specification is defined, we can automatically generate the architectural component view. The workflow for the generation process is depicted in Fig. 1. First, a class model is extracted from the source code. The extraction of a class model from source code decouples our approach from a specific source language since the approach works on language independent UML class models. For instance, for Java different tools exist that can perform this extraction [24,25]. Then, a model transformation is used to generate a UML component view. This model transformation uses the architectural abstraction specification defined in the DSL code and the class model as inputs, and it generates UML component views from this input. The architectural abstraction specification is needed here as it describes the relation between the abstract model and the source code. The model transformation also generates bi-directional traceability information that links the DSL, the class models, and the architectural component views. During the model transformation, consistency checks are applied to identify potential discrepancies between design and code.

As the software system evolves over time, we use our architectural abstraction in the DSL to create multiple architectural component views, e.g. one for each version of the software systems. Our approach can also be used to compare the created models to each other and to architectural component views created manually during early software architecture design by the software architects. This way, software architects can identify where the implementation differs from the original design or from previous versions. They can then argue whether these changes are intended (e.g. flaws in the design) or not (e.g. the implementation is not in accordance with the design). The comparison of these very similar models with only minor differences is a straightforward task. Approaches for advanced model comparison and a variety of frameworks that implement this functionality already exist (see e.g. [26,27]). Based on this comparison, model consistency between a model and consecutive versions can be checked. For example, if the original design model is compared to models that were generated based on the existing implementation, the comparison can indicate which components are not yet implemented or how communication between components works in the current implementation with respect to the intended design.

This approach enables developers to maintain an architecture documentation by providing an “up-to-date” architectural component view that reflects the source code. In order to support the developers in the definition of architectural abstractions and throughout the development our approach generates traceability links between the source model and all generated artifacts. These links provide direct navigation between the different artifacts in our tool.

In addition to the already mentioned consistency checking between different versions of the software, our approach provides automatic consistency checking between the different artifacts of the same version. These checks are based on the automatically generated traceability links and are automatically triggered whenever one of the artifacts is changed. The implemented consistency checks cover the following artifacts: source model, the architectural abstraction specification, and the architectural component view.

If this approach is used in a software development project from the beginning, the software architect drafts an architecture of the system as an abstraction specification that describes components that do not yet exist and that will be implemented over time. This way the architect can keep track of the already implemented components using consistency checks as well as generate an abstraction model whenever she desires to do so. As the architecture of the system evolves, adjustments to the architecture abstraction specification are likely, while non-architectural changes should not have an effect on our architecture abstraction specification.

Our approach also eases another frequently discussed problem in software projects: Often, the connection between design and source code is lost during development. Using our architectural abstraction approach, developers can keep track of which parts of the source code correspond to which architectural components, introducing traceability links from the architectural model to the source code and vice versa.

Multiple architectural abstractions can be defined for the same source code to create different views at different levels of abstraction, where one architectural abstraction provides an overview of a system and other architectural abstractions provide detailed views of different parts of the system on varying levels of abstraction.

Our proof-of-concept implementation[1] uses the EMF implementation of UML [11] for its class and architectural component view. This way it is possible to leverage architectural component views created during design time and repeatedly compare them to the current status of the implementation.

4. Domain specific language for specifying architectural abstractions

To support the architectural abstraction from the automatically created class models to the architectural component views, we define a DSL based on Xtext 2.3 [28]. This DSL provides rules for abstracting the detailed UML classes into architectural components. The rules for defining the abstractions can be grouped into three categories:

- *Name- or ID-based filters*: This category of filters selects classes based on the name or ID of an object; for example all classes that reside in a specific package or all classes that contain the string “message” in their name.
- *Relation-based filters*: These filters select classes based on their relationships to a selected class or interface; for example all classes implementing a specific interface.
- *Compositions*: This category contains set operations instead of actual filters. Using set operations one can manipulate the result sets from other filters in order to combine a number of resource sets or define exclusions from more general filters.

We provide a number of different clauses that map groups of class model elements to components in the architectural component view and to define exceptions to these rules. For the manipulation of sets we provide three basic operations (`union`, `intersection`, and `complement`). For more flexibility, we also added custom filters implemented in Java or Xtend [29]. For this purpose we introduce two special clauses. The Java extension is supported using a filter that is implemented as a static Java method. This method has to accept two parameters: the DSL object of type `JavaExtensionFilter` and a List of `Package` objects. The method is expected to return all UML classifiers that passed the filter. A similar clause exists for using custom filters defined in Xtend.

A complete list of all the clauses that we defined for architectural abstractions can be found in Table 1.

The required and provided interfaces of a component are automatically deduced from the UML-class model by defining all external interfaces, used by the component’s implementing classes, as required interfaces. All interfaces that are implemented by a component’s implementing classes and used by another component are deduced as provided interfaces. The interface details can be hidden by aggregating interfaces in ports as dedicated interaction points and using assembly connectors between the ports.

Additionally our DSL supports the definition of connectors between components. A connector in the architectural abstraction specification supports the definition of realizing objects by using a) the same clauses that are available for components and or b) by specifying a UML dependency relation from the UML class model as the implementing realization. If realizing objects are specified they are stored in the generated UML component view in the form of a UML `Realization` relation that can later be used for consistency checking.

In our examples and studies that we have used to incrementally refine and evaluate the DSL, this set of language elements has been proven to be sufficient to express architectural abstractions in a way that tolerates all kinds of non-architectural changes (see Sections 5.1 and 5.2 for details on five cases of open source projects).

An excerpt of the DSL specification is depicted in Listing 1 as an Xtext grammar. It shows the definition of the infix operators for union (`and`), intersection (`or`), and complement (`and not`). `{,}` can be used to change the operator precedence. The complete Xtext grammar can be found in Appendix A.

The transformation is implemented in Xtend [29] which is first defined for the abstract type `Expression` and then refined for each of the DSL’s clauses.

[1] This prototype is available at https://swa.univie.ac.at/ArchitectureAbstractionDSL.

```
ComponentDef returns ComponentDef:
    'Component' name=ID
    'consists of' (expr=OrComposition)
    connectors+=ConnectorAnnotation*;

ConnectorAnnotation:
    {ConnectorAnnotation}
    'connector to' targets+=[ComponentDef] (',' targets+=[ComponentDef])*
    ('implemented by'
    (implementingExpression+=OrComposition)?
    ('relation: ' implementingRelations+=[umlMM::Dependency|QUALIFIED_NAME]
    (',' implementingRelations+=[umlMM::Dependency|QUALIFIED_NAME])*)?)?
    ;
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 '}';
[...]
```

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

Table 1
Architectural abstraction DSL clauses.

Filter	Parameters	Description
class name	String	all classes, who's name matches the regular expression
package name	String	all classes residing in packages with names matching the regular expression
contained in package	ID	all classes residing in the package identified by the ID
uses	ID	all classes using the class identified by the ID
used by	ID	all classes used by the class identified by the ID
child of	ID	all child classes of the class identified by the ID
super type	ID	all super classes of the class identified by the ID
instance of	ID	all instances of the interface identified by the ID
Java extension	String	String that points to a static Java method which takes the filter object and a List of UML packages as parameters and returns a list of matching classifiers
Xtend extension	String	String that identifies an Xtend function which has the same as the aforementioned Java method.
and	two clauses	infix operator for intersecting the results from two clauses
or	two clauses	infix operator for uniting the results from two clauses
and not	two clauses	infix operator for the difference between two results

4.1. *Illustrative example*

Let us illustrate the use of our architectural abstraction DSL with a simple example. Fig. 2 shows a high-level architectural component view for the Frag project [30]. Frag is a dynamic programming language implemented in Java, specifically designed for supporting building DSLs and supporting MDD. An excerpt of an architectural abstraction specification in the DSL which is used to generate the component view depicted in Fig. 2 can be found in Listing 2.

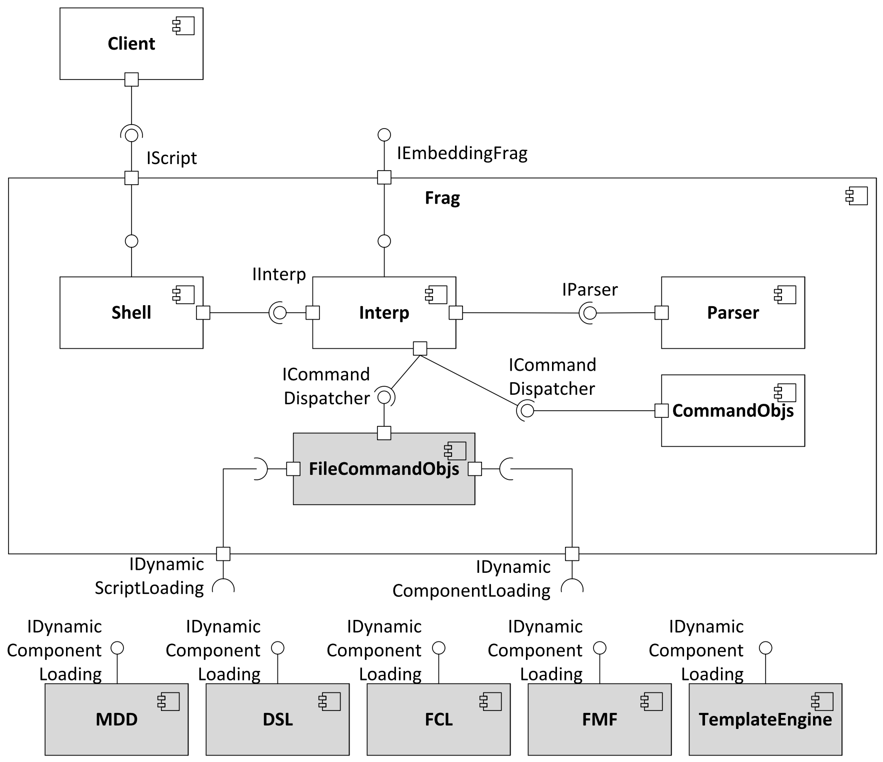


Fig. 2. Visualization of the Frag (v0.91) example for an architectural component view generated from an architectural abstraction specification. Components that were newly introduced between Frag (v0.6) and Frag (v0.9) are colored in gray.

```
Component Interpreter
    consists of {
        Class (".*Interp") or {
            { // all classes the interpreter uses
              // that reside within the core package
                UsedBy (root.frag.core.Interp)
                and Package (root.frag.core)
            } and not Class (root.frag.core.Dual)
        }
    }
    connector to Parser
    connector to CommandObjects
        implemented by Class(root.frag.core.CommandDispatcher)
    connector to FileCommandObjects
        implemented by Class(root.frag.core.CommandDispatcher)
Component CommandObjects
    consists of Package (root.frag.objs)

Component Shell
    consists of
        Class (".*Shell") or {
            {   UsedBy (root.frag.Shell)
                and Package (root.frag.core)
            }
            and not {
                Class (root.frag.core.Interp)
                or Class(root.frag.core.Dual)
            }
        }
    connector to Interpreter
```

Listing 2. Code samples for architectural abstraction of the three main components of the Frag (v0.91) example.

4.2. Automatic generation of traceability links

The generation of traceability links is integrated in the architectural component view transformation. As the transformation is executed, the architectural abstraction specification (i.e., the DSL code) is evaluated for each component. This evaluation yields a set of realizing classes and interfaces for each component. This set is stored in the generated UML component view using a UML Realization relation which is defined by the UML-Standard [11], as a relation between two sets of model elements: The supplier (component) specifies the behavior that is realized by the client (a set of classes and interfaces). This relationship is navigable in both directions and thus able to provide the answers to the question "Which classes and interfaces realize component X?" as well as to the question "Which component does this class (partially) realize?". For example, when looking at the "Interpreter" component shown in Fig. 2 the list of realizing source code elements that are stored in the component realization contains 5 classes (`Interp`, `FragObject`, `FragMethod`, `CodeLine`, `Callframe`).

Using traceability links, our prototype provides navigable links from the architecture specifications to the source model and vice versa. For example, when the user clicks on `root.frag.objs` in Line 4 of Listing 2, the according element in the source model is automatically opened and the clicked element is highlighted.

The traceability links are used in our prototype to provide the developer with a direct navigation from the source model to a generated architectural component view. For instance, when the developer opens the context menu for the `Interp` class in the Eclipse UML2 editor our prototype provides a context-menu entry that, when clicked, opens the according component in architecture abstraction specification as well as the generated component in the generated UML2 component view.

While the automatic generation of traceability links from source code is nothing new, our approach uses them specifically to provide interactive tool support to aid the developers and architects in the definition of the architectural abstractions. In addition, the traceability links are the basis for most of our consistency checks, as explained in the next section.

4.3. Consistency checking during model transformation

Once an architectural abstraction is defined, it is important to identify discrepancies between design and code. To aid this task, our approach supports design constraint checking. Constraints that have to hold for the class model and the architectural component view can be checked, and then discrepancies can be identified by determining which parts of an implementation are not visible in the design and vice versa. At the moment we have implemented checks for the following constraints and plan on implementing further checks in the future:

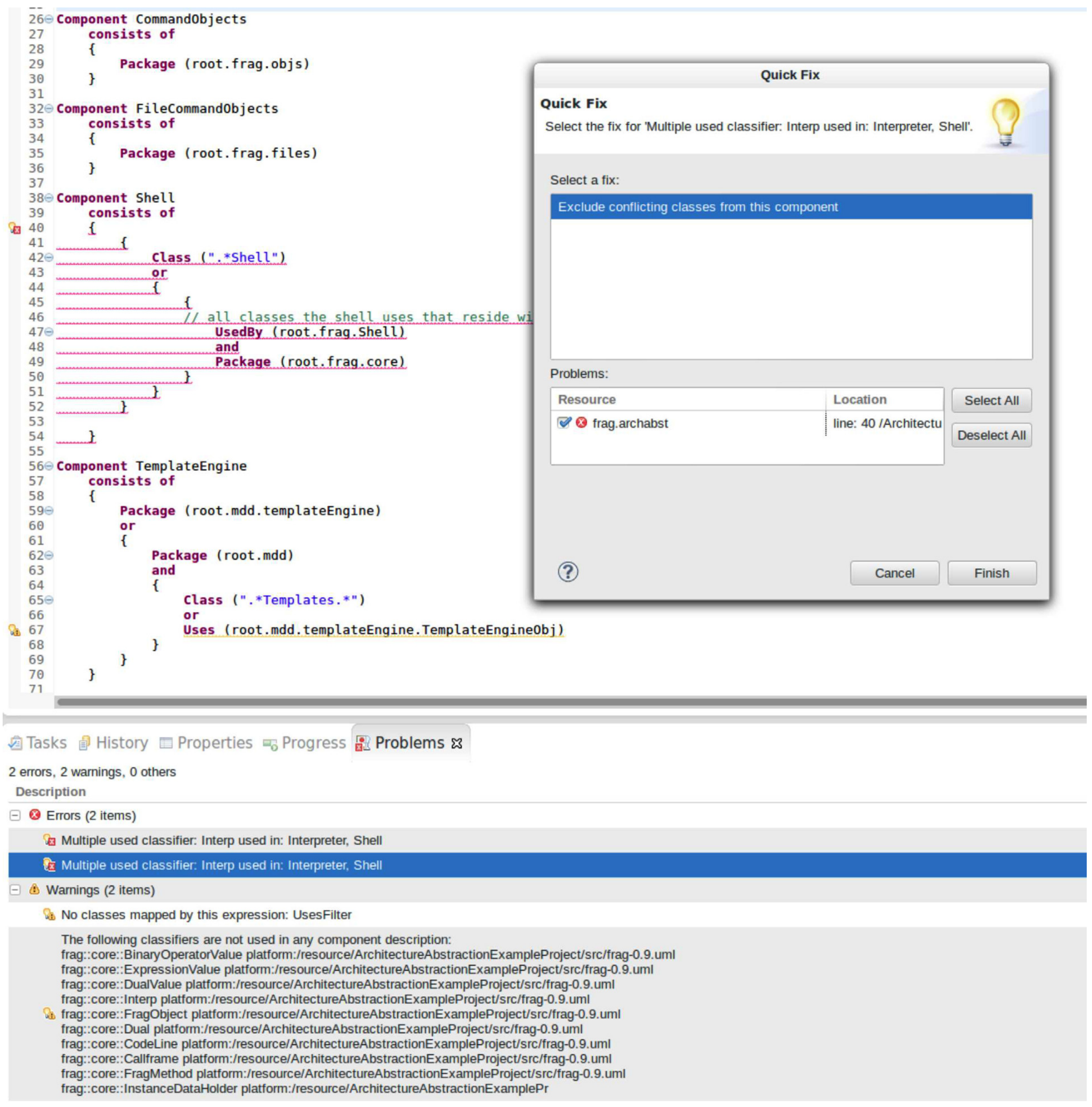
- Mapping of a particular class to multiple components
- Components where the DSL clauses do not map to any realizing elements in the source model
- DSL clauses not matching any classes in the class model
- Particular classes that are not mapped to any component
- Components that have a connector in the architecture abstraction specification but do not have a relation in the source model
- Components that have a relation in the source model but do not have a connector in the architecture abstraction specification. E.g. components that make use of an interface that is defined in another component.

Constraint checking is realized in our prototype using Xtext2's [29] validation framework. This framework distinguishes between *cheap*, *normal*, and *expensive* constraint checks. While *cheap* checks are automatically executed whenever a change in DSL-editor happens, *normal* checks are executed when the model is saved, and *expensive* checks are manually triggered. The majority of our consistency checking constraints are realized as cheap checks, and a few more expensive constraints are realized as normal checks; expensive constraints are not used.

For a number of constraints our prototype automatically provides possible solution options. This ranges from the creation of a new component for classes that are not mapped to any component, over modifying clauses that do not match any source code elements in a way that they possible target a larger number of source code elements, to removing classes that are mapped by multiple components from one of the mapped components.

Fig. 3 shows two exemplary inconsistencies that were reported for the Frag [30] example. The first inconsistency is an error reporting that one or more source code elements where added to more than one component in the architectural abstraction specification. Using the already mentioned traceability links, the consistency checks found that the class `Interp` is contained in the architecture abstraction specifications of the components `Shell` and `Interpreter`. This inconsistency is reported with a marker on every abstraction specification the conflicting class is part of. In Fig. 3 you can also see a proposed solution for this problem: The exclusion of the class `Interp` either from the component `Shell` or the component `Interpreter`.

The other inconsistency reported in Fig. 3 is a warning that shows the lists of elements from the source code that are not part in any of the defined components. This features provides valuable information to the software architect as it provides information about possible starting points for the next iteration in the architecture recovery process. While source code elements that are used in multiple components are reported as errors, unused source code elements are only implemented



```
Component CommandObjects
    consists of
    {
        Package (root.frag.objs)
    }

Component FileCommandObjects
    consists of
    {
        Package (root.frag.files)
    }

Component Shell
    consists of
    {
        {
            Class (".*Shell")
            or
            {
                {
                // all classes the shell uses that reside wi
                    UsedBy (root.frag.Shell)
                    and
                    Package (root.frag.core)
                }
            }
        }

    }

Component TemplateEngine
    consists of
    {
        Package (root.mdd.templateEngine)
        or
        {
            Package (root.mdd)
            and
            {
                Class (".*Templates.*")
                or
                Uses (root.mdd.templateEngine.TemplateEngineObj)
            }
        }
    }
```

Quick Fix

Quick Fix

Select the fix for 'Multiple used classifier: Interp used in: Interpreter, Shell'.

Select a fix:

Exclude conflicting classes from this component

Problems:

Resource	Location
frag.archabst	line: 40 /Architectu

2 errors, 2 warnings, 0 others

Description

Errors (2 items)
- Multiple used classifier: Interp used in: Interpreter, Shell
- Multiple used classifier: Interp used in: Interpreter, Shell

Warnings (2 items)
- No classes mapped by this expression: UsesFilter
- The following classifiers are not used in any component description:

```
frag::core::BinaryOperatorValue platform:/resource/ArchitectureAbstractionExampleProject/src/frag-0.9.uml
frag::core::ExpressionValue platform:/resource/ArchitectureAbstractionExampleProject/src/frag-0.9.uml
frag::core::DualValue platform:/resource/ArchitectureAbstractionExampleProject/src/frag-0.9.uml
frag::core::Interp platform:/resource/ArchitectureAbstractionExampleProject/src/frag-0.9.uml
frag::core::FragObject platform:/resource/ArchitectureAbstractionExampleProject/src/frag-0.9.uml
frag::core::Dual platform:/resource/ArchitectureAbstractionExampleProject/src/frag-0.9.uml
frag::core::CodeLine platform:/resource/ArchitectureAbstractionExampleProject/src/frag-0.9.uml
frag::core::Callframe platform:/resource/ArchitectureAbstractionExampleProject/src/frag-0.9.uml
frag::core::FragMethod platform:/resource/ArchitectureAbstractionExampleProject/src/frag-0.9.uml
frag::core::InstanceDataHolder platform:/resource/ArchitectureAbstractionExamplePr
```

Fig. 3. Exemplary reports of inconsistencies.

as a warning as there might be cases where the software architect does not want to include some source code elements in the architecture abstraction specification.

5. Evaluation

To evaluate our approach with regard to the required effort in creating and maintaining architectural abstractions we conducted five case studies. We have performed these case studies as replications of the same research steps (explained below) for five open source projects of different size and in different application areas. The case studies illustrate how the features of the approach like consistency checks and traceability links are used during an architecture recovery.

As approaches like the one introduced in this paper can potentially require a lot of computational effort, we measured the performance of our tool-suite while creating and maintaining the architectural abstraction specifications for our cases and present the results from this evaluation in Section 5.2.

5.1. Detailed cases of architectural abstraction evolution

In this section we discuss the five open source projects Apache CXF [31], Frag [30], Hibernate [32], Cobertura [33], and Freecol [34] that we have used in our evaluation. During the incremental refinement of our DSL design, we started with scenarios from these projects and extended the set of scenarios step-by-step to cover all changes observed in multiple versions of the five cases studied in this section. The lessons learned from this examples are discussed in Section 6.

We have performed the five case studies as replications of the same research steps: First, we automatically generated a UML class model from the source code using our parser. Then we tried to gain an initial understanding of the program. In order to ease this task we imported the source code in an Eclipse IDE. After an initial study of the source code, we created a first, incomplete architecture abstraction specification. The time that was required to create this initial specification heavily depended on the size and the previously existing architectural knowledge about the example cases. Then we utilized the consistency checks to further improve the abstraction specification by removing the reported inconsistencies step-by-step. The inconsistency at this point usually were source code elements that had not been considered in the abstraction specification.

When we were satisfied with the resulting architecture abstraction specification, we updated the source model to a newer version. After that we checked the architecture specification and the new source model for inconsistencies. Any reported inconsistencies were fixed before we continued with the next version of the program.

For all of the examples, we report the following data points:

- The estimated lines of source code give an impression of the projects size.[2]
- The number of changes in the source code between different versions together with the number of changes in our architecture abstraction specification necessary to account for the changes in the source code indicate how stable our abstractions are.[3] This is directly related to answering RQ1 and indirectly to answering RQ2 and RQ3 as the number of changes between different versions of the examples is related to the number of reported inconsistencies and therefore also to the number of used traceability links.
- The number of components in the architecture abstraction specification as well as the average, standard deviation, and median for the number of classifiers per component provide measures for the abstraction level of our architecture abstraction specification and show that the resulting components are roughly of similar size. This is important, as a single component holding all classes would not have been representative for finding the necessary changes to the architecture abstractions. Therefore this data is related to answering RQ1.
- The time to create and to update the architecture abstraction specifications helps to find an answer to RQ4 as it indicates the amount of effort that is necessary for creating and maintaining architectural abstraction specifications.
- The estimated time required to gather the architectural knowledge indicates the necessary effort if this approached is used for software architecture reconstruction and not from the beginning of a software project.
- The execution time of our prototype for all example cases, which indicates an answer to RQ5, is reported and discussed in Section 5.2.

In the following section we report the data for each example case and discuss the lessons learned in Section 6.1.

5.1.1. Case 1: Apache CXF

Apache CXF is an open source services framework that helps developers build and develop various kinds of Web services. In the Apache CXF case, we used the architecture overview that is available from the CXF web-site[4] as a basis for our source code study which required about 2 hours. After this, the first architectural abstraction specification took 15 minutes to create. However it was not complete and the consistency checks reported a relevant number of source code elements that were not considered in the specification. We then spent another two and half hours studying the source code while incrementally improving our architectural abstraction specification and all the time reducing the number of missing source elements that our consistency checks reported. We continued to the point where the consistency checks did not report any inconsistencies.

Next, we updated the source model to a newer version and adapted our architecture abstraction specification to the changed source model. We repeated the last step until for all versions of Apache CXF. We estimate the total time required for updating the specification to accommodate the changes in the source models with 45 minutes (see Fig. 4).

To show the ability to provide different views for a system we created a detail view for the “Transport” component in the CXF architecture overview (see Fig. 5).

The results in Table 2 show that in order to keep the Apache CXF abstraction up-to-date hardly any changes were necessary. In the course of the evolution of Apache CXF from version 2.0.10 to version 2.4.3 more than 5000 changes were implemented but only ten changes to the architectural abstraction specification were necessary. These modifications

[2] We used the tool `SLOCCount` [35] to estimate the lines of source code.

[3] We used the Linux command line tool `diff` [36] to estimate the changes between the different versions of the cases.

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

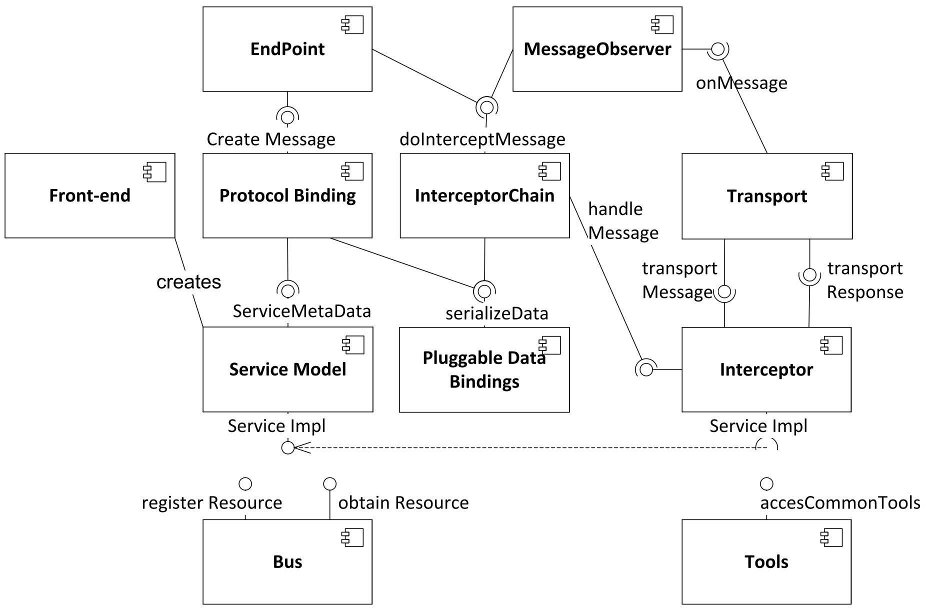


Fig. 4. Apache CXF (v2.4.3) architecture overview [31].

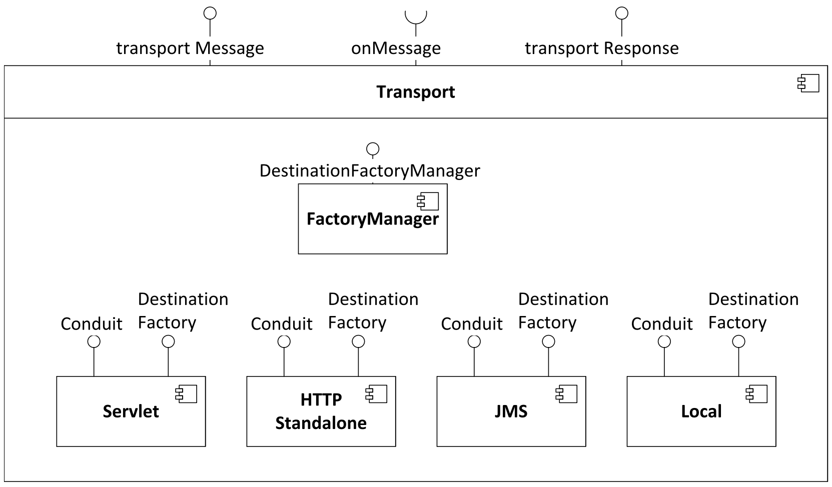


Fig. 5. Detail view for Apache CXF (v2.4.3) transports.

Table 2
Necessary changes to the architectural abstraction specification (DSL code) compared to source changes in Apache CXF.

Apache CXF version	Files added	Files removed	Files changed	Total changes	DSL changes
Overview 2.0.10 ⇒ 2.2.12	299	83	1040	1422	4 new packages; 2 removed packages
Overview 2.2.12 ⇒ 2.3.7	133	19	923	1075	3 new packages
Overview 2.3.7 ⇒ 2.4.3	115	62	739	916	1 new package
Transport 2.0.10 ⇒ 2.2.12	29	4	90	123	1 new component
Transport 2.2.12 ⇒ 2.3.7	17	3	117	137	1 new component
Transport 2.3.7 ⇒ 2.4.3	20	23	120	163	1 component removed

constitute eight new and two removed packages that were introduced between the different versions. This result might be caused by the fact that we only compared minor revisions (no older version than Apache CXF 2.0.10 is available) during which no major changes to the architecture were made.

When looking at the detail view for the transport component in Table 2, three changes were necessary. The package “http_osgi” was added in version 2.2 and removed in version 2.4, and the package “jaxws_http_spi” that was added in version 2.3.

Table 3
Apache CXF: Average, median, and standard deviation (σ) for the number of classes per component.

CXF version	sLoC	# of components	Avg. # of classifiers per component	σ for the # of classifiers	Median for the # of classifiers
Overview 2.0	198k	11	123.64	93.04	98.00
Overview 2.2	342k	11	180.18	150.15	159.00
Overview 2.3	370k	11	194.23	154.11	185.00
Overview 2.4	390k	11	212.73	174.22	193.00
Transport 2.0	1196	5	26.40	14.69	25.00
Transport 2.2	1399	5	31.40	16.89	25.00
Transport 2.3	1715	5	38.20	20.10	30.00
Transport 2.4	1229	5	38.00	22.03	30.00

Table 4
Necessary changes to the architectural abstraction specification (DSL code) compared to source changes in Frag.

Frag version	Files added	Files removed	Files changed	Total changes	DSL changes
0.6 ⇒ 0.7	48	44	32	124	2 new components, 2 minor changes
0.7 ⇒ 0.8	9	1	148	158	2 new components, 6 minor changes
0.8 ⇒ 0.91	7	40	36	83	no changes

Table 5
Frag: Average, median, and standard deviation (σ) for the number of classes per component.

Frag version	sLoC	# of components	Avg. # of classifiers per component	σ for the # of classifiers	Median for the # of classifiers
0.6	10k	7	21.57	25.20	6.00
0.7	12k	10	16.00	23.83	6.00
0.8	14k	11	18.73	23.93	10.00
0.9.1	13k	11	14.18	11.29	11.00

Table 3 contains additional data for this case study. It shows the average number of classifiers per component, the standard deviation for the number of classifiers per component, the median number of classifiers per component, and the lines of source code (in thousands) for the specific version of the program. The standard deviation indicates that the components in the CXF case vary significantly in size (number of classifiers). While in CXF 2.4 the smallest component has only 36 realizing classifiers, the largest component is realized by 534 classifiers. However as the median indicates, the size of the different components is dispersed between these extremes.

The details for the Transport view are also summarized in Table 3. As can be seen 5 components with an average of 26.4–38 classifiers have been introduced, again with substantial deviations between different components.

5.1.2. Case 2: Frag

Frag is a dynamic programming language implemented in Java, specifically designed for supporting building DSLs and supporting MDD. The high-level architecture of Frag in Version 0.91 was shown already in Fig. 2.

One of the authors of the paper is also the author of Frag and thus could provide a UML component diagram within half an hour. Using this architectural information as a starting point, the effort required to create an architecture abstraction specification for Frag took about 15 minutes while updating the specification to the newer versions took about 40 minutes.

During this evolution of Frag's architecture, we identified a number of differences when comparing the architecture of Version 0.91 to the architecture of Version 0.6, which is missing the components DSL, FCL, FMF, and TemplateEngine. Fig. 2 highlights these differences. Components that were part of Frag 0.6 have a white background, while components that where introduced in the newer versions have a gray background.

The architectural abstraction specification for Frag 0.6 has less than fifty lines of DSL code and shows a very straightforward architecture. The changes necessary to conform to Frag 0.7 are shown in Listing 3. They constitute a substantial modification to the architectural abstraction specification. This was expected, since in this revision the architecture of Frag had been reworked to use the Java Reflection API for dynamic dispatching of Frag method calls. Also a number of new features were introduced that led to new components. These components were grouped in a new package called `mdsd`.

Prior to these changes, our consistency checks reported 3 missing packages and 85 classes that were not considered in the architectural abstraction specification for Frag.

For the following version of Frag (0.8) 7 inconsistencies were reported by our prototype. Another new component (TemplateEngine) was introduced which required twelve lines of DSL code and the top-level package `mdsd` was renamed to `mdd`, which required updates to the architectural abstraction specification at six places that were automatically highlighted by the consistency checks. The integration of partial support for such automatic architectural abstraction specification updates are a topic for future research.

```
Component Parser
    consists of {
    Package(root.frag.parser)
-   and not
-   Package(root.frag.parser.predefinedObjs)
    }
Component CommandObjects
    consists of {
+   Package(root.frag.objs)
-   Package(root.frag.parser.predefinedObjs)
-   or Package(root.frag.predefinedObjects)
    }
+Component DSL
+   consists of {
+       Package(root.mdsd.dsl) or {
+           Package(root.mdsd) and Class(".*DSL.*")
+       }
+   }
+Component FCL
+   consists of {
+       Package(root.mdsd.fcl) or {
+           Package(root.mdsd) and Class(".*FCL.*")
+       }
+   }
+Component FMF
+   consists of
+       Package(root.mdsd) and Class(".*FMF.*")
```

Listing 3. Architectural abstraction specification modifications for the changes in Frag 0.7.

Another change was that the code for the FMF component was moved into a package of its own, with only one class remaining outside this package. These changes account for 5 new lines of DSL code.

For the following release (Frag 0.91) the number of changes halved and no changes to the architecture were made. Because of this, no inconsistencies are reported and no updates to the architectural abstraction specification are required. A summary of all the changes that occurred during the evolution of Frag is shown in Table 4.

The data on how classifiers are distributed to components in this case can be found in Table 5. As the average and standard deviation suggest, the number of classes per component varies. However as the median is close to the average, the size of the components is evenly distributed between the minimum and maximum.

5.1.3. Case 3: Cobertura

Cobertura is a Java code coverage analysis tool that can be used to determine what percentage of your source code is exercised by a unit test suite. For Cobertura, we created an initial architectural abstraction specification on the basis of Version 1.0 through source code study.

With no prior architectural knowledge available, the effort to conduct the source code study and create an initial architectural abstraction specification was roughly 40 minutes and another 15 minutes of improving this specification. The total required time to adapt the architecture abstraction specification from Cobertura 1.0 step by step until Cobertura 1.9.4.1 was about 70 minutes.

A simplified version of the architecture of Cobertura is depicted in Fig. 6.

Table 6 summarizes the evolution of architectural abstraction specification until the most recent Version 1.9.4.1. While initial versions only contained about 4000 lines of source code, Version 1.9.4, the last available version, has about 50 000 lines of source code.

As Table 7 shows, the last version only has three more components than the initial version. All three introduced components were predated by a reported inconsistency that listed newly introduced classes and the only other reported inconsistency was a package that was removed in version 1.3 and thus could no longer be used in the architecture abstraction specification.

While the components are of similar size from Version 1.0 through to Version 1.9, in Version 1.9.4 the `JavaNCSS` component triples in size. This is caused by the new package `net.sourceforge.cobertura.javancss.parser`. As a result, the standard deviation increases from 3.24 to 14.06. This new parser might be a candidate for a component in its own right. However, as this parser is only used within `JavaNCSS`, we decided against interpreting this as an architectural change, as this parser is an internal implementation detail of the `JavaNCSS` component and in our opinion not relevant to the overall architecture of Cobertura. This is why we accepted the increased standard deviation and kept the, compared to the other components, large `JavaNCSS` component.

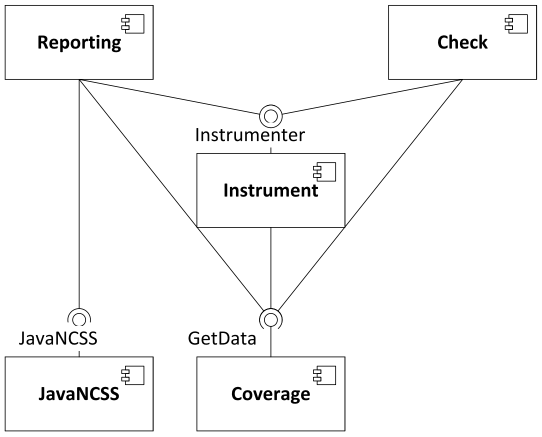


Fig. 6. Simplified architecture overview of Cobertura 1.1.

Table 6
Necessary changes to the architectural abstraction specification (DSL code) compared to source changes in Cobertura.

Cobertura version	Files added	Files removed	Files changed	Total changes	DSL changes
1.0 ⇒ 1.1	18	10	23	51	no changes
1.1 ⇒ 1.2	23	1	2	26	no changes
1.2 ⇒ 1.3	12	5	8	25	1 new component and 1 minor change
1.3 ⇒ 1.4	25	3	3	31	no changes
1.4 ⇒ 1.5	20	2	0	22	no changes
1.5 ⇒ 1.6	17	3	1	21	no changes
1.6 ⇒ 1.7	18	1	0	19	no changes
1.7 ⇒ 1.8	47	11	1	59	1 new component
1.8 ⇒ 1.9	16	10	1	27	no changes
1.9 ⇒ 1.9.4	26	14	9	49	1 new component
1.9.4 ⇒ 1.9.4.1	1	0	0	1	no changes

Table 7
Cobertura: Average, median, and standard deviation for the number of classes per component.

Cobertura version	sLoC	# of components	Avg. # of classifiers per component	σ for the # of classifiers	Median for the # of classifiers
1.0	4239	5	11	7.66	10.00
1.1	3491	5	8	5.87	7.00
1.2	3421	5	8	5.81	7.00
1.3	3406	6	5	2.61	4.50
1.4	3556	6	6	2.95	4.50
1.5	4003	6	6	2.77	5.50
1.6	3956	6	7	2.51	6.50
1.7	3997	6	7	2.17	6.50
1.8	17 165	7	7.86	2.67	9.00
1.9	18 179	7	9.14	3.24	10.00
1.9.4	51 343	8	13.25	14.06	10.50
1.9.4.1	51 342	8	17	17.10	10.50

5.1.4. Case 4: Hibernate

Hibernate is an open source Java persistence framework that supports object-relational mapping and querying of databases.

Although we found some information about the Hibernate architecture in the projects documentation, we still required a substantial amount of time for the initial source code study, which required about 3 hours. We then required 30 minutes for the creation of the initial architecture abstraction specification, another 150 minutes for improving the specification, and after that about 75 minutes for updating the specification according to the newer versions of Hibernate.

A simplified version of the architecture of Hibernate is depicted in Fig. 7 and Table 8 presents the changes we encountered over the different versions. Especially interesting are the changes from version 3.6.10 to version 4.0.0alpha1 and from this version to 4.0.0.final, since these two changes constitute the transition between two major versions. While this version's changes consisted of a huge number of modified files and more than 150 new classes, only eight inconsistencies were reported which led to eight minor changes (added or removed packages) in the DSL code but no added or removed

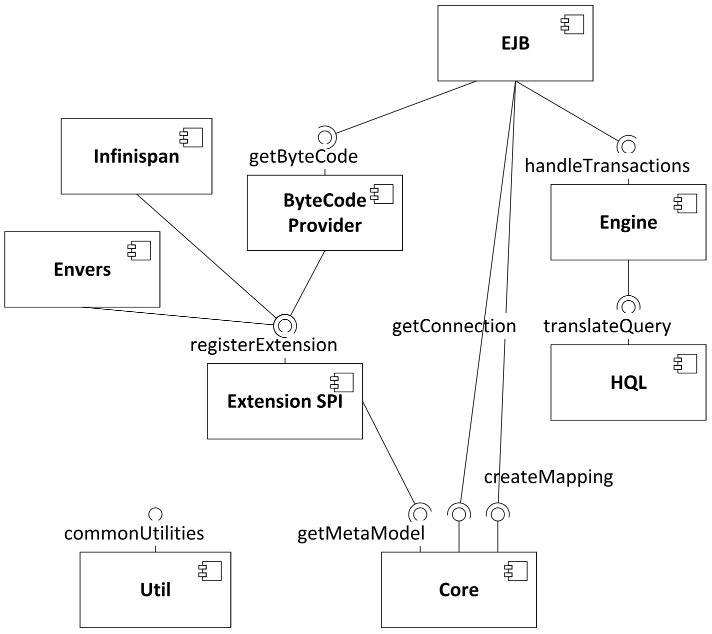


Fig. 7. Simplified architecture overview of Hibernate 4.1.10.

Table 8
Necessary changes to the architectural abstraction specification (DSL code) compared to source changes in Hibernate.

Hibernate version	Files added	Files removed	Files changed	Total changes	DSL changes
3.6.6 ⇒ 3.6.10	22	0	89	111	no changes
3.6.10 ⇒ 4.0.0.alpha1	20	86	2597	2703	3 removed packages; 3 new packages
4.0.0.alpha1 ⇒ 4.0.0.final	174	420	1270	1864	2 removed packages
4.0.0 ⇒ 4.0.1	19	2	70	91	no changes
4.0.1 ⇒ 4.1.0	54	5	221	280	no changes
4.1.0 ⇒ 4.1.1	12	5	192	209	no changes
4.1.1 ⇒ 4.1.2	15	5	720	740	no changes
4.1.2. ⇒ 4.1.3	20	3	391	414	no changes
4.1.3 ⇒ 4.1.4	12	2	87	101	no changes
4.1.4 ⇒ 4.1.5	132	1	92	225	no changes
4.1.5 ⇒ 4.1.6	22	0	87	109	no changes
4.1.6 ⇒ 4.1.7	8	0	54	62	no changes
4.1.7 ⇒ 4.1.8	34	11	147	192	1 new package
4.1.8 ⇒ 4.1.9	16	1	118	135	no changes
4.1.9 ⇒ 4.1.10	42	1	112	155	no changes

component. In our opinion, a part of the reason for this low level of change is the high level of abstraction that is used in the architectural component view for this case.

As the standard deviation reported in Table 9 indicates, the components in this case are of varying size. The smallest component is realized by approx. 40 classifiers while the three largest components have more than 500 realizing classifiers.

5.1.5. *Case 5: Freecol*

Freecol is an open source game implemented in Java that is based on the popular game Colonization. While Version 0.4 has only 30 000 lines of code, the last version we examined has more than 100 000 lines of code.

We found no existing architectural information for FreeCol. This is why the necessary source code study required about 3.5 hours before we could create a first architectural abstraction specification. Updating the specification to the newer versions of FreeCol then required another 60 minutes.

The results for this example are presented in Table 10 and an architecture overview is shown in Fig. 8.

Although the size of this project more than tripled, no inconsistencies were reported by our prototype and therefore no changes to the architectural abstraction specification (the DSL code) were necessary. The main cause for this stable architecture abstraction is that most changes to the game were improvements to the graphical user interface, improvements to the gameplay, or bug fixes.

Table 11 presents the data about component sizes for this case. The high averages of classes per component (compared to the medians) indicate that this architectural abstraction consists of a number of small components and a very few larger

Table 9
Hibernate: Average, median, and standard deviation (σ) for the number of classes per component.

Hibernate version	sLoC	# of components	Avg. # of classifiers per component	σ for the # of classifiers	Median for the # of classifiers
3.6.6	344k	9	225.89	219.91	174.00
3.6.10	351k	9	226.78	220.60	174.00
4.0.0.alpha1	343k	9	293.33	240.14	210.00
4.0.0.final	387k	9	298.67	261.32	189.00
4.0.1	389k	9	295.00	256.85	184.00
4.1.0	400k	9	296.33	258.13	184.00
4.1.1	402k	9	296.22	257.96	185.00
4.1.2	405k	9	301.11	262.16	191.00
4.1.3	407k	9	297.11	257.24	186.00
4.1.4	409k	9	297.22	257.48	186.00
4.1.5	410k	9	302.44	263.30	192.00
4.1.6	413k	9	298.11	258.78	187.00
4.1.7	416k	9	298.11	258.78	187.00
4.1.8	421k	9	306.22	249.67	189.00
4.1.9	423k	9	306.33	249.63	189.00
4.1.10	426k	9	308.22	252.77	190.00

Table 10
Necessary changes to the architectural abstraction specification (DSL code) compared to source changes in Freecol.

Freecol version	Files added	Files removed	Files changed	Total changes	DSL changes
0.4 ⇒ 0.5.0	190	55	0	245	no changes
0.5.0 ⇒ 0.5.3	155	9	1	165	no changes
0.5.3 ⇒ 0.6.0	185	72	1	258	no changes
0.6.0 ⇒ 0.6.1	60	6	0	66	no changes
0.6.1 ⇒ 0.7.0	217	31	0	248	no changes
0.7.0 ⇒ 0.7.4	356	31	1	388	no changes
0.7.4 ⇒ 0.8.0	284	86	18	388	no changes
0.8.0 ⇒ 0.8.4	117	10	0	127	no changes
0.8.4 ⇒ 0.9.0	280	72	14	366	no changes
0.9.0 ⇒ 0.9.5	370	18	18	406	no changes
0.9.5 ⇒ 0.10.0	468	95	71	634	no changes
0.10.0 ⇒ 0.10.7	542	88	22	652	no changes

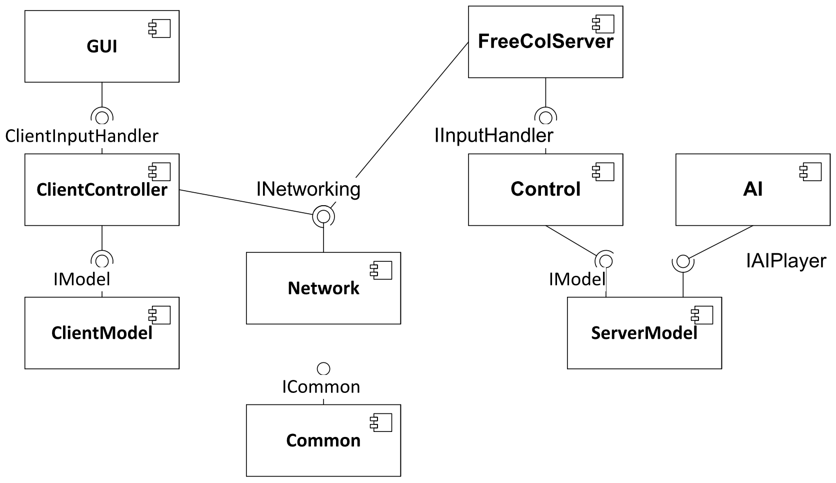


Fig. 8. Simplified architecture overview of FreeCol 0.10.7.

components. In this case the large components are `GUI` and `Common`. While the purpose of the first component is rather self-explanatory, the component “Common” provides functionality that is used on the client and on the server. That is, it provides functions that the client and server have in common. We focused our architectural abstraction on the “big picture” architecture of the system to convey an understanding for the whole system.

Table 11
Freecol: Lines of java source code (in thousands), average, median, and standard deviation (σ) for the number of classes per component.

Freecol version	sLoC	# of components	Avg. # of classifiers per component	σ for the # of classifiers	Median for the # of classifiers
0.4	31k	9	25.44	40.51	9.00
0.5	42k	9	28.00	44.00	9.00
0.5.3	47k	9	28.22	44.20	9.00
0.6	52k	9	33.67	57.08	9.00
0.6.1	54k	9	37.56	61.42	9.00
0.7.0	60k	9	41.44	67.81	9.00
0.7.4	63k	9	42.56	70.48	9.00
0.8.0	74k	9	51.22	83.90	9.00
0.8.4	76k	9	52.00	84.83	9.00
0.9.0	84k	9	57.89	94.07	10.00
0.9.5	82k	9	56.89	91.57	10.00
0.10.0	92k	9	64.44	103.43	11.00
0.10.7	101k	9	73.44	119.64	12.00

Table 12
Execution times, standard deviation (σ) and other key data for implemented cases.

Project	# Clauses	Avg. exec. time (in ms)	σ (in ms)	Median exec. time (in ms)
Cobertura 1.9.4.1	19	176	257	116
Frag 0.9.1	41	283	281	219
FreeCol 0.9.5	21	704	647	455
Apache CXF 2.4	35	3010	1534	2353
Hibernate 3.6.6	96	3117	956	2757

5.2. Performance evaluation

To validate our approach, we realized architectural abstraction specifications for the five open source projects explained in the previous section (see Table 12). As approaches like the one introduced in this paper can potentially require a lot of computational effort, we measured the performance of our prototype tool-suite for the transformations in the five open source projects. Performance problems can be introduced for instance through the exponential growth of the execution time of the analysis according to the size of the model and the architectural abstraction specification. However, for regular usage of the approach an execution time below two minutes is acceptable. We measured the time it takes to execute the constraint checks and the transformations. Table 12 shows the execution times for the most recent version of each of the five open source cases which we obtained on a developer notebook (Intel i7 L620, 4 Gb RAM). We measured each execution time 100 times and calculated the average value. We also measured the minimum and maximum values, but as we observed only small deviations around the average values, so we only report the averages here.

The results from Table 12 suggest that the execution time increases with the number of clauses in the architectural abstraction specification and with the number of classes in the source code. The results also suggest that the approach is well applicable for even larger projects like Hibernate in the normal flow of software development. Thus we can state that for the example cases it is possible to generate an architectural component view with traceability links and check all artifacts for consistency within an acceptable amount of time (RQ5).

Please note that we did not report the time that is needed for extracting the class model from the source code, since this algorithm converts every class in the source code into an instance in the model. That is, this algorithm has a time requirement of $O(n)$, where n is the number of classes in the project.

6. Discussion

In this section we discuss the lessons learned from the five example cases from the case study as well as the results of the performance evaluation of our prototype. Furthermore we discuss the limitations of the approach and open issues in Section 6.2.

6.1. Lessons learned

The cases discussed in Section 5.1 confirm that it is possible to create abstractions based on the generic filters defined in our DSL (RQ1). In all cases, from version to version a large number of changes have been applied, often in many different files. Still only small changes to the DSL were necessary, even for major architectural changes in the projects.

The times required to create an initial architectural abstraction specification differ between the systems where architectural knowledge was already available in other forms (Apache CXF, Frag) and the systems where we had no or only very limited architectural knowledge about the system prior to our case study (Cobertura, Hibernate, FreeCol). In our opinion, the time that was necessary to come from the initial specification to a satisfactory one depends on factors like the quality of existing architectural information and system size. For instance, while we could create an initial specification for Apache CXF based on an existing box diagram within 15 minutes, it took us long to create a specification that we were satisfied with. For Frag, on the other hand, we had first hand architectural knowledge as one of the authors is also involved in the development of Frag. Thus the creation of a satisfactory architecture abstraction specification took only 20 minutes before we could start comparing the different versions of Frag. Regarding RQ4 we can conclude that the effort of creating a suitable architectural abstraction specification varies heavily depending on the existing knowledge of the source code and architecture. If it exists, a small architectural abstraction specification can be created in about 15–20 minutes.

The consistency checking rules are an important means to automatically indicate that changes to the DSL might be necessary. In our finalized version, all necessary changes between different versions of the examples were automatically detected by the consistency checking. This reduced the necessity for manually searching for changes or the use of other tools like *diff*. In addition the inconsistencies are reported directly for the violating parts of the specification and thus direct the software architect to the origin of the problem at hand. The inconsistency that was reported the most while creating the example cases, and thus was the most helpful, were source code elements that were not considered in the architectural abstraction specification. In summary, in the example cases the consistency checks were very helpful in creating and maintaining architectural abstraction specifications (RQ3).

Traceability links are an important aid to find and understand the links between components and realizing classifiers between two version quickly. That is, without support through consistency checking and traceability links, in our 5 cases, the same low number of updates would have been necessary in our DSL, but the amount of manual searching and understanding of change impacts would have been substantially higher. We used them throughout all the examples to navigate between the architectural abstraction specification and the source code elements whenever an inconsistency was reported. The traceability links provide the foundation for all consistency checks and are especially helpful when source code elements are not considered in the architecture abstraction specification. So with respect to RQ2 we can state that: For these five exemplary cases we are able to generate traceability links that were useful during the definition of architectural abstractions.

Once the architectural abstraction specification is defined, we are able to automatically create abstraction models from the source code. We noticed many of our component definitions are based on one to five *Package* rules. Packages are a major way of grouping multiple Java classes (besides Tagging interfaces and so on). The advantage of component definitions based on existing groupings like packages is that the architectural abstraction specifications can cope with many kinds of changes, as in an established software project the coarse grained (package) structure usually is stable. For this reason, only major changes require a change of the architectural abstraction specification. For example, the introduction of a new subpackage or a new class do not require any changes. Only the introduction of new major packages or new components requires architectural abstraction specification updates. However, while in our use cases the grouping based on *Package* rules was beneficial, this might not be always the case.

Our approach supports the creation of architectural abstraction specifications on different levels of abstraction. The data in Table 12 supports this claim. While we needed 41 clauses to map the 13k lines of code from Frag, we only needed 21 clauses to map the 103k lines of code from FreeCol and 35 clauses for mapping the 386k lines of code of Apache CXF to components. This indicates that the Apache CXF architectural abstraction specification is on a higher abstraction level than the one for Frag.

The five cases indicate that creating and maintaining architectural abstractions is easier for high-level abstractions and that generic filters like package-based or name-based filters are less likely to be changed. For example, name-based filters are unaffected by changes as long as the regular expression for matching the name is not affected. A `Package` rule that uses a regular expression like ".*model.*" only is affected if this exact part of the name is modified, while a `Package`-rule based on the fully qualified name of the package needs to be updated as soon as one of the packages on its path is modified.

However, it is not always possible to define architectural abstraction specifications solely using name-based filters like `Package` name filters and the union of their results (`or`). One example is the Shell component in Listing 2 that consists of all the classes that contain the name "Shell" and all elements in the `root.frag.core` package that are used by `root.frag.Shell`. The definitions based on the relationships between classes have two disadvantages: Firstly, they are hard to read because it is unclear which classes match the specified filter. Secondly, relationship-based filters can have side-effects. A related class can reside within a package that is also targeted by a package-based or a name-based filter. This can be avoided by defining an exception in one of the filters. The evolution of relationship-based filters is similar to the already mentioned `Package`-filters that is based on fully qualified names. They need to be updated only if the class that is defined in the filter is moved, renamed, or deleted.

In our future work, we will evaluate different options for extending our DSL. That is, by adding support for constants that allow the reuse of Strings in architecture abstraction specifications as well as ways to allow the manual definition of component interfaces. This would allow us to define constraints that test whether a component's implementation exposes or uses other interfaces than the ones defined in the architecture abstraction specification.

When we compare the statistical data for the different versions of the cases to the corresponding number of lines of source code, one can see that they develop in a similar manner. As the lines of code grow, the average size of classifiers and the median for the number of classifiers per component also grow. Because of this, we think that the added source code gets distributed among the different components. This negates the potential threat to validity that in our cases, all classes could be aggregated in a single component. In this case obviously no changes to the architectural abstraction would ever be necessary. As already mentioned, this is not the case in any of the presented cases.

6.2. Limitations and open issues

Our approach has limitations when being applied to architectural knowledge recovery and no prior knowledge about a software project exists. Under these circumstances our approach is only applicable after initial architectural knowledge has been acquired, since it does not provide an automated abstraction that can be used for refinement. This limitation does not reduce the applicability in a software development project where the focus lies on preserving architectural knowledge. In such cases, the required knowledge usually is created in an early stage of a software development project (i.e. this problem will not arise in the first place).

While we demonstrated our approach in this paper on cases that use programming languages supporting the structuring of source code (e.g., via packages in Java), our approach is also applicable for other languages that do not offer such features. The limitation that arises from the missing structuring of source code features is that the `Package` filter cannot be used. All other rules are still available and can be used instead. However, this limitation often increases the number of rules necessary to define an architectural abstraction specification, though.

In our five case studies, we documented the architectures on a high abstraction level. It is likely that a more detailed architectural descriptions of the same example cases lead to more changes during the evolution of the systems. This is partly reflected in the Apache CXF Transport case. However the main goal of our approach is to support the understanding of the “big picture”. This is the reason why we chose a high abstraction level for our case studies. We would like to investigate in future research in how far our approach is applicable for detailed design as well.

Furthermore, while our approach is designed to be independent of the source programming language by using UML as the representation of the source model and a number of tools exist for different programming languages that transform from source code into UML2 model elements, we currently use examples that are implemented in Java. The reason for this is that we currently only have implemented a transformation from Java to our input representation in EMF UML2.

7. Related work

In this section we compare to related approaches that either use similar techniques or try to solve similar problems. Fig. 9 gives an overview of the approaches that we briefly present in this section. We have split the related work into different groups: In Subsection 7.1 we present a number of selected articles that apply different approaches making use of automatic clustering. Subsection 7.2 discusses articles that propose different kinds of model-based approaches that create abstractions or views from source code. Subsection 7.3 presents works that focus on model evolution and consistency checking of models. The next Subsection 7.4 compares to approaches that focus on traceability and/or change impact analysis. Finally, Subsection 7.5 presents selected approaches that are either hybrid approaches or other approaches that do not fit into one of the other sections but are nonetheless relevant for our work.

Many of the approaches discussed in this section are related to the field of software architecture reconstruction. Ducasse and Pollet [37] presented a survey on the state-of-the-art in the field of software architecture reconstruction. They analyzed and categorized the existing approaches with respect to their goals, inputs, process, techniques, and outputs.

7.1. Automatic clustering approaches

Abreu et al. introduce a reengineering approach using cluster analysis [6]. This approach uses six different affinity schemes and seven clustering methods to produce a series of clustering proposals to verify which one produces the best results. In contrast to our approach, the clustering leads to solutions similar to those proposed by human experts only if the average number of classes per module is not too high.

Another approach for recovering architecture information is introduced by von Detten and Becker [8]. The authors combine clustering and (anti-)pattern information to extract components from existing source code. This work has a different focus than our approach: While both approaches abstract from low-level model representations of a software project, we introduce an extra step of defining the architectural abstraction specification in the DSL, which removes the uncertainty of using automatic clustering approaches and provides the software designer with more control.

Corazza et al. [9] introduce a clustering approach that uses lexical information. It uses a probabilistic model and the Expectation Maximization algorithm to weigh this information and customizes the K-Medoids algorithm in order to group classes. In their case study they compare their approach with other automatic clustering approaches previously compared by Bittencourt and Guerrero [38]. As already mentioned in Section 1, the case study by Corazza et al. [9] states that the authoritativness values are close to 0.5 in 5 of 7 cases. This means that in five cases, it is necessary to execute move or join operations for about half the entities. Our approach removes the necessity to correct the automated clustering but requires the effort to maintain the architectural abstraction specifications.

Maletic and Marcus [39] used an automatic clustering approach that utilized latent semantic indexing for the data-retrieval and a minimal spanning tree for partitioning the data. This approach shares the same problem with aforementioned clustering approaches: The results it produces need to be manually corrected. We believe that our approach creates less

	Approach	Focus	Method	Input	Level of automation	Evolution support
Clustering approaches	Abreu et al.	Reengineering	Cluster analysis	Source code only	Automatic	No
	von Detten and Becker	Architecture Recovery	Cluster analysis	Anti pattern information	Automatic	No
	Corazza et al.	Architecture Recovery	Cluster analysis	Lexical information	Automatic	No
	Maletic and Marcus	Architecture Recovery	Cluster analysis	Latent semantic indexing, minimal spanning tree	Automatic	No
	Dietrich et al.	Architecture Recovery	Cluster analysis	Dependency graphs	Automatic	No
	De Lucia et al.	Tracelink recovery	Cluster analysis	Latent semantic indexing	Automatic	No
Model-based approaches	Scanniello et al.	Layers detection	Graph-based model	ER graph, architecture patten graph	Semi-automatic	No
	Ivkovic and Kontogiannis	Model synchronization	Graph-based model	Transformation model	Automatic	No
	Egyed	Model abstraction	UML model	Traceability information, abstraction rules	Automatic	No
	Brosig et al.	Component model extraction	Palladio component model	EJBs and their runtime control flows	Automatic	No
	Murphy et al.	Mapping source models to high-level models	Model transformation	Low-level model, high-level model, mapping	Automatic	Yes
	Hassan and Holt	Software Architecture Reconstruction	Model transformation	Same as Murphy et al., version control modification records	Automatic	Yes
	Mens et al.	Source code views	Logic programming rules	Logic programming rules	Semi-automatic	No
	Pinzger et al.	Architecture analysis	Graph-based model	Problem reports, change logs	Automatic	No
	Richner and Ducasse	Reverse Engineering	Logic facts	Prolog queries	Manual	No
	Riva and Rodriguez	Software Architecture Reconstruction	Graph-based model	Static and dynamic information	Manual	No
Model Evolution and consistency approaches	Sabetzadeh et al.	Global consistency checking	Model merge	Multiple models, consistency constraints	Semi-automatic	Yes
	Ajila and Alam	Model evolution	Checks model modifications for consistency	Constraints	Semi-automatic	Yes
	Sangal et al.	Conformance checking	Dependency-structure matrices	Source code only	Automatic	Yes
	Moor et al. (.QL)	Conformance checking	Source code queries	Source code queries	Manual	Yes
	Feilkas et al.	Measuring the loss of architectural knowledge	XML	Mapping (based on regular expressions)	Semi-automatic	Yes
	Steyaert et al.	Conformance checking	Reuse contracts	Interface definitions annotated with reuse information	Semi-automatic	Yes
	Our approach	Architecture documentation & evolution	EMF/Xtext	Architecture abstraction specification	Semi-automatic	Yes

Fig. 9. Overview of selected related work.

maintenance effort because no manual corrections are necessary. This is based on the fact that manual corrections are needed after every execution of a clustering algorithm, while our architectural abstraction specification does not create additional effort for multiple applications of the approach.

Dietrich et al. [7] describe an approach for analyzing Java dependency graphs with clustering. However this approach still needs the configuration of the separation level (the number of iterations of removing the edges with the maximum betweenness level). While our approach does not work fully automatically, it allows several versions of a model that can be incrementally fine-tuned by the user. Our approach also provides stable results when changes in the code are made.

De Lucia et al. [40] integrate a latent semantic indexing approach [41] into a software artifact management system in order to recover traceability links. However they also state that one of the limitations in using information retrieval techniques is that in order to find all traceability links, it is necessary to manually discard a big amount of false positives.

All approaches discussed so far deal with automatic recovery of design knowledge. More clustering approaches and clustering measures are reviewed and compared by Maqbool and Babri [42]. They define a number of groups of clustering algorithms and compare the performance of the different groups for different open source software projects. While Maqbool and Babri conclude which approach works best for each of the applications, they do not draw any conclusions regarding the overall effort necessary to correct the automatic clustering. A lot of clustering approaches assume that no architectural knowledge about a system exists. In contrast to all these approaches, our approach is semi-automatic, enables the checking of design constraints during the abstraction process, and provides traceability between source code and models.

7.2. Model-based abstraction and views

Various approaches have been proposed for creating abstractions or views from source code. Scanniello et al. [43] propose an approach for semi-automatically detecting layers in software systems based on the algorithm introduced by Kleinberg [44]. The authors implemented a prototype and provide a case study for JHotDraw.[5] While their approach is fo-

[5] http://www.jhotdraw.org/.

cused on semi-automatically detecting layers without prior knowledge, our approach is focused on supporting the evolution of the program and its architecture by providing abstractions on different levels.

Sartipi describes a pattern-based approach for recovering software architecture [45]. 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 architectural component views are only used for consistency checks.

Ivkovic and Kontogiannis [46] provide an approach for keeping models synchronized. However, they base their approach on an additional graph-based meta-model and a transformation model for synchronization. In contrast, our approach makes it easier to trace the corresponding low-level objects in the source code, since no intermediary models are needed.

Egyed [17] describes an approach for model abstraction by using existing traceability information and abstraction rules. However, the author identified 120 abstraction rules for the example of UML class models, which need to be extended with a probability value because the rules may not always be valid. Our approach uses architectural abstraction specifications that are harder to reuse but easier to define and allow the definition of architectural abstraction specifications on different levels of abstraction.

Brosig et al. [47] describe how they extract a Palladio component model from Enterprise Java Beans. However, their approach is based on EJBs and the runtime control flow while our approach is not limited to EJBs and based on statically analyzing the existing source code.

Another approach for mapping source code models to high-level models is introduced by Murphy et al. [16]. They use software reflexion models which they compute from a mapping between source model and high-level model. However, while their approach is similar, it requires a substantial amount of effort, since it requires to define both: the high-level model and the mapping, while our approach requires source code and architectural abstraction specification and the architecture abstraction is generated automatically.

An interesting software architecture recovery approach is introduced by Hassan and Hol [48]. It uses modification records from source code versioning systems. While the approach provides additional information to the developers, it basic workings do not differ from the already discussed approach from Murphy et al. [49].

Mens et al. [15] propose intentional source code views that allow grouping of source code by concerns. These views are defined in a logic programming language. Their approach provides generic source views on a low abstraction level while we focus on the architectural aspects and provide an easy way to define our domain specific views.

An approach that integrates source code with information found in problem reports and changelogs to do architecture analysis is presented by Pinzger et al. [50]. The analysis produces specific directed attributed graphs which then are integrated into a FAMIX [51] model and compute architectural views using binary relational algebra based on an approach introduced by Holt et al. [52]. These views then show intended and unintended couplings between architecture elements. While our approach also allows the abstraction of higher level views on a system, it does not make use of an intermediate graph representation which allows us to support traceability between abstraction specification and source model.

Richner and Ducasse [53] introduce an approach for reverse engineering that models the static and dynamic aspect of an object oriented application as logic facts. Then an engineer creates abstractions based on these facts and Prolog queries using these abstractions generate high-level views. These are generated using clustering rules in Prolog. While this approach has similarities with our presented approach, we designed our approach specifically with the evolution of a software system in mind. Furthermore, our approach defines architecture abstraction directly on a source model which allows us to use consistency checking and supports traceability between the abstraction and the source model.

Another graph-based approach for architecture reconstruction is introduced by Riva and Vidal Rodriguez [54]. They create architectural abstraction using static and dynamic information about a software system using a Prolog system. Like most of the architecture reconstruction however, it does not take the evolution of a software artifact into account, while our approach specifically targets this evolution and tries to prevent architectural knowledge evaporation.

7.3. Model evolution and consistency

In this subsection, we present different existing approaches that focus on model evolution and consistency checking of models. Sabetzadeh et al. [55] describe an approach for consistency checking through model merge. While consistency checking is a part of our work, we mainly focus on the architectural abstraction specification and providing additional value for projects that do not use model driven development per se. Furthermore, we focus on models that provide different levels of abstraction while the model merge approach is better applicable to models on the same abstraction level.

Ajila and Alam describe a formal approach for model evolution by extending OMGs Object Constraint Language [56] with "Constraint with Action Language" [57]. It uses annotated directed acyclic graphs as model representations and works directly on the single model and its modifications. However, our approach is targeted at creating an abstraction in the form of an architectural component view from source code and keeping track of the changes is done implicitly by only comparing the different versions of the abstract model.

Passos et al. [58] give an illustrative overview on static architecture-conformance checking. They compare three approaches: The Lattix Dependency Manager (LDM) [59], which is based on Dependency-Structure Matrices, .QL [60], which is a source code query languages (SCQL) [61,62], and the reflexion models (RM) introduced by Murphy et al. [16]. As Passos et al. summarize, all of these approaches have drawbacks. While the LDM tool has very limited capabilities of expressing constraints, .QL has only a low abstraction level, and RMs have only limited support for architecture reasoning and discovery. Our approach features an expressive DSL that can provide abstraction on different levels and is capable of automatically generating abstractions from the architectural abstraction specifications.

Feilkas et al. [63] perform an industrial case study on the loss of architectural knowledge during system evolution. To measure the loss of architectural knowledge, they use an approach based on machine readable component descriptions and policies in XML. Their approach offers only limited ways to describe mappings between components and source code. Their mappings are solely based on regular-expressions that map package-names to components. As our case studies show this is not always sufficient to describe components and our approach provides more flexible ways to describe architectural abstraction specifications.

Steyaert et al. [64] introduce Reuse Contracts which are interface descriptions, created in addition to a software artifact, that describe additional design information. The authors introduce their approach exemplary for inner-class relationships but claim that it is easily adaptable for inter-class relationships. However, our approach has a different focus. While Steyaert et al. mainly focus on reuse, we aim to support the evolution of a software product by providing a simple way to create up-to-date views on different abstraction levels.

Knodel et al. [65] give an overview on how and when static architecture evaluations can contribute to architecture development and show how architecture development is influenced by architecture evaluations in the area of software product lines. While our approach targets the maintenance and recovery of architecture information in general, the approach introduced by Knodel et al. [65] focuses on evaluating a system's architecture regarding software product lines. While they use a source model and a high level model and require a mapping created by a human to compare these two models, our approach uses the source model and the mapping is a starting point, while the high level abstraction model is generated automatically.

7.4. Approaches that focus on traceability and/or change impact analysis

Feng and Maletic [66] present an approach to analyze the impact of changing components at runtime based on slicing on component interaction traces. Their dynamic component composition model is based on a static model and a set of UML sequence diagrams. While their approach focuses on the analysis of the impact of dynamic changes during runtime, our approach primarily focuses on providing architectural abstractions from source code to architectural component views during software evolution and thus focuses on changes at compile time between different evolution steps of a software system.

Another approach that focuses on change impact analysis is presented by Zhao et al. [67]. They present an automated approach that uses architectural slicing and chopping to analyze formal architectural specifications based on WRIGHT [68]. While this approach uses an architectural specification as an input, our approach is focused on semi-automatically providing this architecture abstraction. Thus it might be interesting to integrate both approaches in order to provide change impact analysis on an architectural level that is able to also provide knowledge about the changes effects on the source code of the system.

A wide variety of work has been done in the field of traceability: i.e. traceability of concerns between architectural views [69], to linking design decisions to design models [70]. Winkler and Pilgrim [71] present a survey on the state of the art of traceability in requirements engineering and model-driven development who base their work amongst others on the works by Spanoudakis and Zisman [72] who, based on extensive literature study, define eight different kinds of traceability links, discuss a number of approaches for the generation and maintenance of traceability links, and present a number of ways how traceability relations can be used in software development.

7.5. Hybrid and other approaches

Harris et al. [73] propose a style library and use a recognition engine to detect instances of these abstractions in the source code. Guo et al. [74] 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 [75]. As all automatic approaches, these approaches require manual correction of false positives and cannot guarantee a hundred percent detection rate.

Lungu et al. [76] 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.

Qingshan et al. [77] describe an architecture recovery approach that focuses on extracting the Process Structure Graph (PSG) of a system. While this approach works fully automatic, it does not allow to create views on different levels of abstraction but is limited to the PSG while our approach allows to define different levels of abstraction while requiring the manual creation of architecture abstraction specifications.

8. Conclusion

In this paper, we presented a semi-automatic approach for supporting architectural abstractions of source code into architectural component views. Our approach, supported through a DSL and MDD tooling, can automatically generate architectural component views from source code and supports traceability between the mapped artifacts. By creating architectural abstraction specifications with the DSL on different levels of abstraction, we are able to generate different abstracted views for one project. A major feature of our approach is its ability to cope with change. Only major changes, like newly introduced components, require an update of the architectural abstraction specification. Overall, our evaluations and experience show that architectural abstraction specifications can be created and maintained with low effort. This is due in part to the traceability links and inconsistency checking support. That is, we can check for inconsistencies between abstraction and code and identify the participating source code and architectural components in case constraints are violated.

Our approach has limitations when used for reengineering as knowledge about the source code and the design of a project are needed to create the architectural abstraction specification; in many reengineering approaches the main assumption is that such knowledge does not yet exist. Hence, our approach can be used together with these approaches: The reengineering approaches can be used for acquiring an understanding of a project, and our approach can be used to maintain and evolve an architectural view on the system once it has been sufficiently understood. We plan to investigate this relation in our future work.

In our future work we also plan to increase the usability of the presented approach by implementing support for other popular programming languages by providing the necessary transformations from source code to the UML2 class model. For this purpose we will focus on languages that are already supported in the Eclipse IDE like C++.

Furthermore we plan to extend this approach towards the architectural pattern discovery and documentation. For this purpose we will allow the developer to annotate components and connectors from the architecture abstraction specification with architectural primitives [78] and then search pattern candidates that are built of these architectural primitives.

Appendix A. Xtext grammar of the architecture abstraction DSL

The source code for our proof-of-concept implementation can be found at https://git.swa.univie.ac.at/component-model/component-model-source (only for registered users). It is available under the MIT license. Listing 4 shows the complete Xtext grammar of our architectural 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

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

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

ComponentDef returns ComponentDef:
    'Component' name=ID
    'consists of' (expr=OrComposition)
    connectors+=ConnectorAnnotation*;

ConnectorAnnotation:
    {ConnectorAnnotation}
    'connector to' targets+=[ComponentDef] (',' targets+=[ComponentDef])*
    ('implemented by' (implementingExpression+=OrComposition)?
    ('relation: ' implementingRelations+=[umlMM::Dependency|QUALIFIED_NAME]
    (',' implementingRelations+=[umlMM::Dependency|QUALIFIED_NAME])*)?)?;

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')? ')';

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

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

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

ExtensionFilter:
    JavaExtensionFilter | XtendExtensionFilter;

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

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

Listing 4. Complete Xtext grammar of our architectural abstraction DSL.

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, p. 4.

[2] S. Brown, Software Architecture for Developers, Leanpub, Vancouver, BC, Canada, 2013.

[3] R. Koschke, Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey, J. Softw. Maint. 15 (2) (2003) 87–109, http://dx.doi.org/10.1002/smr.270.

[4] D. Sun, K. Wong, On evaluating the layout of uml class diagrams for program comprehension, in: Proceedings of the 13th International Workshop on Program Comprehension, IWPC 2005, 2005, pp. 317–326.

[5] M. Eiglsperger, M. Kaufmann, M. Siebenhaller, A topology-shape-metrics approach for the automatic layout of uml class diagrams, in: Proceedings of the 2003 ACM Symposium on Software Visualization, SoftVis '03, ACM, New York, NY, USA, 2003, p. 189.

[6] F.B.e. Abreu, G. Pereira, P. Sousa, A coupling-guided cluster analysis approach to reengineer the modularity of object-oriented systems, in: Proceedings of the Conference on Software Maintenance and Reengineering, CSMR '00, IEEE Computer Society, Washington, DC, USA, 2000, p. 13, http://dl.acm.org/citation.cfm?id=518900.795263.

[7] J. Dietrich, V. Yakovlev, C. McCartin, G. Jenson, M. Duchrow, Cluster analysis of Java dependency graphs, in: Proceedings of the 4th ACM Symposium on Software Visualization, SoftVis '08, ACM, New York, NY, USA, 2008, pp. 91–94.

[8] M. von Detten, S. Becker, Combining clustering and pattern detection for the reengineering of component-based software systems, in: Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS, QoSA-ISARCS '11, ACM, New York, NY, USA, 2011, pp. 23–32.

[9] A. Corazza, S. Di Martino, G. Scanniello, A probabilistic based approach towards software system clustering, in: Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering, CSMR '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 88–96.

[10] N. Rozanski, E. Woods, Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Addison-Wesley Professional, 2005.
[11] Object Management Group, UML 2.3 superstructure, http://www.omg.org/spec/UML/2.3, 2010.
[12] N. Medvidovic, R.N. Taylor, A classification and comparison framework for software architecture description languages, IEEE Trans. Softw. Eng. 26 (2000) 70–93, http://dx.doi.org/10.1109/32.825767, http://dl.acm.org/citation.cfm?id=331520.331527.
[13] O. Zimmermann, U. Zdun, T. Gschwind, F. Leymann, Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method, in: Seventh Working IEEE/IFIP Conference on Software Architecture, WICSA 2008, 2008, pp. 157–166.
[14] 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.
[15] K. Mens, T. Mens, M. Wermelinger, Maintaining software through intentional source-code views, in: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, SEKE '02, ACM, New York, NY, USA, 2002, pp. 289–296.
[16] G.C. Murphy, D. Notkin, K. Sullivan, Software reflexion models: bridging the gap between source and high-level models, in: Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT '95, ACM, New York, NY, USA, 1995, pp. 18–28.
[17] A. Egyed, Consistent adaptation and evolution of class diagrams during refinement, in: Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004, ETAPS 2004, Barcelona, Spain, in: Lecture Notes in Computer Science, vol. 2984, Springer, 2004, pp. 37–53.
[18] M. Fowler, Domain-Specific Languages (Addison-Wesley Signature Series (Fowler)), 1st edition, Addison-Wesley Professional, 2010.
[19] R.C. Gronback, Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit, 1st edition, Addison-Wesley Professional, 2009.
[20] K.-K. Lau, Z. Wang, Software component models, IEEE Trans. Softw. Eng. 33 (10) (2007) 709–724, http://dx.doi.org/10.1109/TSE.2007.70726.
[21] J. Ivers, P. Clements, D. Garlan, R. Nord, B. Schmerl, J.R.O. Silva, Documenting component and connector views with uml 2.0, Tech. Rep. CMU/SEI-2004-TR-008, Software Engineering Institute (Carnegie Mellon University), 2004.
[22] S. Beydeda, V. Gruhn, Model-Driven Software Development, Springer-Verlag, New York, Inc., Secaucus, NJ, USA, 2005.
[23] A.R. Hevner, S.T. March, J. Park, S. Ram, Design science in information systems research, MIS Q. 28 (1) (2004) 75–105.
[24] D. Spinellis, UMLGraph, http://www.umlgraph.org, 2011.
[25] Soyatec, eUML2, http://www.soyatec.com/euml2/, 2011.
[26] Eclipse Foundation, EMF compare, http://www.eclipse.org/emf/compare/, 2011.
[27] M. Kofman, E. Perjons, Metadiff – a model comparison framework, metadiff.sourceforge.net/docs/metadiff.pdf.
[28] Eclipse Foundation, Xtext, http://www.eclipse.org/Xtext/, 2011.
[29] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, M. Hanus, Xbase: implementing domain-specific languages for java, in: Proceedings of the 11th International Conference on Generative Programming and Component Engineering, GPCE '12, ACM, New York, NY, USA, 2012, pp. 112–121.
[30] U. Zdun, The Frag language, http://frag.sourceforge.net/, 2011.
[31] Apache CXF, http://cxf.apache.org, 2011.
[32] J. Linwood, D. Minter, Beginning Hibernate, 2nd edition, Apress, Berkely, CA, USA, 2010.
[33] M. Doliner, J. Erdfelt, J. Lewis, G. Lukasik, J. Mareš, J. Thomerson, Cobertura, http://cobertura.sourceforge.net, 2011.
[34] The Freecol Team, Freecol, http://freecol.org, 2011.
[35] D. Wheeler, Sloccount, http://www.dwheeler.com/sloccount/, 2009.
[36] J.W. Hunt, M.D. McIlroy, An algorithm for differential file comparison, Tech. Rep. CSTR 41, Bell Laboratories, Murray Hill, NJ, 1976.
[37] S. Ducasse, D. Pollet, Software architecture reconstruction: A process-oriented taxonomy, IEEE Trans. Softw. Eng. 35 (4) (2009) 573–591.
[38] R.A. Bittencourt, D.D.S. Guerrero, Comparison of graph clustering algorithms for recovering software architecture module views, in: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA, 2009, pp. 251–254.
[39] J.I. Maletic, A. Marcus, Supporting program comprehension using semantic and structural information, in: Proceedings of the 23rd International Conference on Software Engineering, ICSE '01, IEEE Computer Society, Washington, DC, USA, 2001, pp. 103–112, http://dl.acm.org/citation.cfm?id=381473.381484.
[40] A. De Lucia, F. Fasano, R. Oliveto, G. Tortora, Recovering traceability links in software artifact management systems using information retrieval methods, ACM Trans. Softw. Eng. Methodol. 16 (4) (September 2007), Article 13, http://dx.doi.org/10.1145/1276933.1276934.
[41] S.C. Deerwester, S.T. Dumais, T.K. Landauer, G.W. Furnas, R.A. Harshman, Indexing by latent semantic analysis, J. Am. Soc. Inf. Sci. 41 (1990) 391–407.
[42] O. Maqbool, H. Babri, Hierarchical clustering for software architecture recovery, IEEE Trans. Softw. Eng. 33 (2007) 759–780, http://dx.doi.org/10.1109/TSE.2007.70732.
[43] 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.
[44] J.M. Kleinberg, Authoritative sources in a hyperlinked environment, J. ACM 46 (1999) 604–632, http://dx.doi.org/10.1145/324133.324140.
[45] K. Sartipi, Software architecture recovery based on pattern matching, in: Proceedings of the International Conference on Software Maintenance, ICSM '03, IEEE Computer Society, Washington, DC, USA, 2003, p. 293, http://dl.acm.org/citation.cfm?id=942800.943546.
[46] I. Ivkovic, K. Kontogiannis, Tracing evolution changes of software artifacts through model synchronization, in: Proceedings of the 20th IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, 2004, pp. 252–261, http://dl.acm.org/citation.cfm?id=1018431.1021433.
[47] F. Brosig, S. Kounev, K. Krogmann, Automated extraction of palladio component models from running enterprise Java applications, in: Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS '09, ICST, Brussels, Belgium, 2009, pp. 10:1–10, http://dx.doi.org/10.4108/ICST.VALUETOOLS2009.7981.
[48] A.E. Hassan, R.C. Holt, Using development history sticky notes to understand software architecture, in: Proceedings of the 12th IEEE International Workshop on Program Comprehension, IWPC '04, IEEE Computer Society, Washington, DC, USA, 2004, p. 183.
[49] G.C. Murphy, D. Notkin, K. Sullivan, Software reflexion models: bridging the gap between source and high-level models, in: Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT '95, ACM, New York, NY, USA, 1995, pp. 18–28.
[50] M. Pinzger, H. Gall, M. Fischer, Towards an integrated view on architecture and its evolution, Electron. Notes Theor. Comput. Sci. 127 (3) (2005) 183–196.
[51] S. Tichelaar, S. Ducasse, S. Demeyer, Famix and xmi, in: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00), WCRE '00, IEEE Computer Society, Washington, DC, USA, 2000, p. 296.
[52] R.C. Holt, Structural manipulations of software architecture using tarski relational algebra, in: Proceedings of the Working Conference on Reverse Engineering (WCRE'98), WCRE '98, IEEE Computer Society, Washington, DC, USA, 1998, p. 210, http://dl.acm.org/citation.cfm?id=832305.837013.
[53] T. Richner, S. Ducasse, Recovering high-level views of object-oriented applications from static and dynamic information, in: Proceedings of the IEEE International Conference on Software Maintenance, ICSM '99, IEEE Computer Society, Washington, DC, USA, 1999, p. 13.
[54] C. Riva, J.V. Rodriguez, Combining static and dynamic views for architecture reconstruction, in: Proceedings of the Sixth European Conference on Software Maintenance and Reengineering, CSMR '02, IEEE Computer Society, Washington, DC, USA, 2002, p. 47.
[55] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook, M. Chechik, Consistency checking of conceptual models via model merging, in: 15th IEEE International Requirements Engineering Conference, 2007, RE '07, 2007, pp. 221–230, http://dx.doi.org/10.1109/RE.2007.18.

[56] Object Management Group, OCL 2.2 specification, http://www.omg.org/spec/OCL/2.2, 2010.
[57] S.A. Ajila, S. Alam, Using a formal language constructs for software model evolution, in: Proceedings of the 2009 IEEE International Conference on Semantic Computing, ICSC '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 390–395.
[58] L. Passos, R. Terra, M.T. Valente, R. Diniz, N. das Chagas Mendonca, Static architecture-conformance checking: An illustrative overview, IEEE Softw. 27 (2010) 82–89, http://dx.doi.org/10.1109/MS.2009.117.
[59] N. Sangal, E. Jordan, V. Sinha, D. Jackson, Using dependency models to manage complex software architecture, in: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05, ACM, New York, NY, USA, 2005, pp. 167–176.
[60] O. Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, J. Tibble, .QL: Object-oriented queries made easy, in: Generative and Transformational Techniques in Software Engineering II, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 78–133.
[61] R.-G. Urma, A. Mycroft, Programming language evolution via source code query languages, in: Proceedings of the ACM 4th Annual Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU '12, ACM, New York, NY, USA, 2012, pp. 35–38.
[62] E. Hajiyev, M. Verbaere, O. de Moor, Codequest: scalable source code queries with datalog, in: Proceedings of the 20th European Conference on Object-Oriented Programming, ECOOP '06, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 2–27.
[63] M. Feilkas, D. Ratiu, E. Jurgens, The loss of architectural knowledge during system evolution: An industrial case study, in: IEEE 17th International Conference on Program Comprehension, ICPC '09, 2009, pp. 188–197.
[64] P. Steyaert, C. Lucas, K. Mens, T. D'Hondt, Reuse contracts: managing the evolution of reusable assets, in: Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM, New York, NY, USA, 1996, pp. 268–285.
[65] J. Knodel, D. Muthig, M. Naab, M. Lindvall, Static evaluation of software architectures, in: European Conference on Software Maintenance and Reengineering, 2006, pp. 279–294.
[66] T. Feng, J.I. Maletic, Applying dynamic change impact analysis in component-based architecture design, in: Proceedings of the Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, SNPD-SAWN '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 43–48.
[67] J. Zhao, H. Yang, L. Xiang, B. Xu, Change impact analysis to support architectural evolution, J. Softw. Maint. 14 (5) (2002) 317–333, http://dx.doi.org/10.1002/smr.258.
[68] R. Allen, D. Garlan, A formal basis for architectural connection, ACM Trans. Softw. Eng. Methodol. 6 (3) (1997) 213–249, http://dx.doi.org/10.1145/258077.258078.
[69] B. Tekinerdogan, C. Hofmann, M. Aksit, Modeling traceability of concerns for synchronizing architectural views, J. Object Technol. 6 (7) (2007) 7–25, http://doc.utwente.nl/60276/.
[70] P. Könemann, O. Zimmermann, Linking design decisions to design models in model-based software development, in: Proceedings of the 4th European Conference on Software Architecture, ECSA'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 246–262, http://dl.acm.org/citation.cfm?id=1887899.1887920.
[71] S. Winkler, J. Pilgrim, A survey of traceability in requirements engineering and model-driven development, Softw. Syst. Model. 9 (4) (2010) 529–565, http://dx.doi.org/10.1007/s10270-009-0145-0.
[72] G. Spanoudakis, A. Zisman, Software traceability: a roadmap, in: S.K. Chang (Ed.), Handbook of Software Engineering and Knowledge Engineering, vol. 3, World Scientific Publishing Co., 2005.
[73] 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.
[74] 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, 1999, pp. 15–34.
[75] R. Kazman, S.J. Carrière, Playing detective: Reconstructing software architecture from available evidence, Automated Softw. Engg. 6 (2) (1999) 107–138.
[76] 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.
[77] L. Qingshan, C. Hua, H. Shengming, C. Ping, Z. Yun, Architecture recovery and abstraction from the perspective of processes, in: Proceedings of the 12th Working Conference on Reverse Engineering, WCRE '05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 57–66.
[78] U. Zdun, P. Avgeriou, A catalog of architectural primitives for modeling architectural patterns, Inf. Softw. Technol. 50 (9–10) (2008) 1003–1034, http://dx.doi.org/10.1016/j.infsof.2007.09.003.