DSL-based Support for Semi-Automated Architectural Component Model Abstraction Throughout the Software Lifecycle

Thomas Haitzer and Uwe Zdun
Software Architecture Group
Faculty of Computer Science
University of Vienna
{thomas.haitzer | uwe.zdun}@univie.ac.at

ABSTRACT

In this paper we present an approach for supporting the semi-automated abstraction of architectural models throughout the software lifecycle. It addresses the problem that the design and the 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 architectural knowledge from implemented systems and keeping the abstracted architectural knowledge up-to-date. 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. The DSL and its tools support abstracting the source code into UML component models for describing the architecture. Once the software architect has defined an architectural abstraction in the DSL, we can automatically generate UML component models from the source code and check whether the architectural design constraints are fulfilled by the models. Our approach supports full traceability between source code elements and architectural abstractions, and allows software architects to compare different versions of the generated UML component model with each other. We evaluate our research results by studying the evolution of architectural abstractions in different consecutive versions and the execution times for five existing open source systems.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; D.2.2 [**Software Engineering**]: Design Tools and Techniques

Keywords

DSL, Architectural Abstraction, Architectural Component Models, Software Evolution, UML, Model Transformation

1. INTRODUCTION

In many software projects the design and the implementation drift apart during development and system evolution [18]. In some small projects this problem can be avoided, as it might be possible to understand and maintain a well written source code without additional architectural documentation. For many systems beyond the hundred thousands or millions lines of code (LoC), 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.

Automatically generated diagrams of the systems (e.g. in form of class diagrams) usually do not represent higher-level abstractions and hence they hardly support the understanding of the big picture. Clustering approaches from the reengineering research literature (e.g. [2, 10, 39]) can help to obtain an initial understanding and make sense of such diagrams, however the case study presented by Corazza et al. [7] 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.

That is the reason why today it is usually necessary to have an up-to-date documentation of the system's architecture that is maintained manually. To model architectural knowledge, often models in UML [31], ADLs [27], or similar modeling approaches are used. Such models often are created before the actual implementation begins and later, during the 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 [41]. This problem is known as *architectural knowledge evaporation* [18].

In this paper, we introduce an approach that focuses on architectural abstractions from the source code in a changing environment while still supporting traceability. While a number of works exist that focus on abstractions from source code [28, 29, 39, 13], so far none of these approaches targets architectural abstractions at different levels of granularity, traceability between architectural models and the code, or the ability to cope with the constant evolution of software systems. Our approach introduces the semi-automatic abstraction of UML component models from the source code based on an architectural abstraction specified in a domain specific language (DSL) [15, 16]. In contrast to the related works, our approach specifically targets architectural abstractions and requires changes to the DSL code 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).

We chose a semi-automatic approach to enable the user 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 was to let the user 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, thus supporting views ranging from high-level software architectural views to more low-level software design views.

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 DSL code. Only changes to the architecture itself, which usually require a substantial modification of source code, require the architectural abstraction specification to be updated. By creating different versions of the component model over time, we are able to use a delta comparison to check how and where the component model has changed. 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.

Once the architectural component models 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. Traceability can be ensured if model-driven development (MDD) [4] is used to generate the code from the models, but MDD is not used in the majority of projects today and by default model-to-text transformations do not create traceability information. This results in additional manual effort needed to identify which code elements are created from which model elements. With our architectural abstraction approach, in contrast, the task of identifying all source code elements that contribute to a specific component is trivial, as we can compute trace links directly from DSL code.

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 illustrate our approach by discussing examples based on the open-source projects Apache CXF [1] and Frag [40]. In Section 6 we discuss lessons learned and the evaluation of our research results using five cases from existing open source systems of varying size. Section 7 compares our approach to the related work. We conclude in Section 8.

2. RESEARCH PROBLEM AND RESEARCH METHOD

During the software development lifecycle, the source code and the architecture of a system evolve and change. This often results in architectural knowledge evaporation [18]. 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 [41]. Additionally, when using conventional means for architecture documentation like abstract 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 [28, 29, 39, 13]. 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 DSL code 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 DSL code.

To reach this goal we have designed and implemented the DSL using an incremental refinement process, following the action research method [8, 20]. Action research is an iterative process where the researchers diagnose a problem and then plan actions to solve the diagnosed problem. The planned actions are then executed and the solution is evaluated. The evaluation results are used to learn from this iteration of the cycle. These lessons learned are used as the starting point for a new iteration of the cycle.

In particular, 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 DSL code.

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 6, the vast majority of changes are non-architectural changes, and they can be tolerated by the architectural abstractions defined in our DSL without requiring changes to the DSL code. Only when changes that are classified as architectural changes are introduce in the open source systems, updates to the architectural abstraction code in the DSL are necessary.

3. APPROACH OVERVIEW

The approach introduced in this paper supports architectural abstractions of a software system under development. It also allows 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 defined a DSL that defines architectural abstractions from class models, which can be automatically extracted from the source code, into archi-

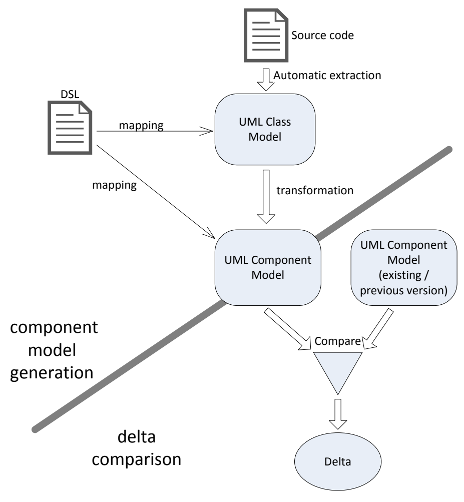

Figure 1: Generating component models from source code and comparing different model versions

tectural component models. If specified appropriately, this architectural abstraction should be 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).

Once an architectural abstraction specification is defined, we can automatically generate the architectural component model. The workflow for the generation process is depicted in Figure 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 [37, 36]. Then a transformation is used to generate a UML-component model. This transformation uses the architectural abstraction specification, defined in the DSL code, and the source code as input and generates the UML component model. The architectural abstraction specification is needed here as it describes the relation between the abstract model and the source code.

We use our architectural abstraction in the DSL to create multiple instances of component models, as well as to compare the created models to each other and to component model instances created at design time. This way, one can identify where the implementation differs from the original design and can then argue whether these changes are intended (e.g. flaws in the design) or not (e.g. developers not sticking to 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. [11, 21]). Based on this comparison, model consistency between a design model and the implementation can be checked. For example, the comparison indicates which components are not implemented and 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" component model that reflects the source code.

Our approach also eases another often discussed problem in software projects: The connection between design and source code often 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 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 uses the EMF implementation of UML [31] for its class and component model. This way it is possible to leverage component models 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 models, we define a DSL based on Xtext [12] which provides rules for abstracting the detailed UML classes into architectural models (UML packages to UML components). These rules can be grouped into three categories: 1. *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. 2. *Relation-based filters*: These filters select classes based on their relationships to a selected class. For example all classes implementing a specific interface. 3. *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 instances in the component model and to define exceptions to these rules. For the manipulation of sets we provide the three basic operations that are relevant for our use case (`union`, `intersection`, and `complement`). For more flexibility, we also added custom filters implemented in Java or Xtend. 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 the DSL can be found in Table 1. 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 and 6 for details on these examples and studies).

An excerpt of the DSL specification is depicted in Figure 2. It shows the definition of the infix operators for union (`and`),

```
Component returns ComponentDef:
'Component(' name=STRING ')'
  '{'
    (expr=OrComposition)
  '}';

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 '}';
```

Figure 2: Excerpt of the Xtext source code of our architectural abstraction DSL

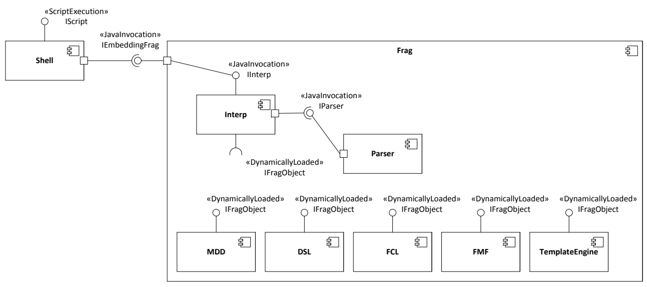

Figure 3: Visualization of the Frag example for a component model generated from an architectural abstraction specification

intersection (`or`), and complement (`and not`). `{,}` can be used to change the operator precedence. The transformation is implemented as an Xtend function, which is first defined for the abstract type `Expression` and then refined for each of the DSL's clauses.

Let us illustrate the use of our architectural abstraction DSL with a simple example. Figure 3 shows a high-level architectural component model for the Frag project [40]. An excerpt of an architectural abstraction specification in the DSL which is used to generate the component model depicted in Figure 3 can be found in Figure 4.

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 during the transformation. Constraints that have to hold for the class model and the component model 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:

```
Component("Parser") {
  Package(root.frag.parser)
}

Component("Shell") {
  Class(".*Shell") or {
    UsedBy(root.frag.Shell) and
    Package(root.frag.core)
  }
}

Component("Interpreter") {
  Class(".*Interp")
  or {
    UsedBy(root.frag.core.Interp) and
    Package(root.frag.core)
  }
}
```

Figure 4: Code samples for architectural abstraction of the three main components of the Frag example

- Mapping of a class to multiple components
- Missing interface implementations
- Unimplemented components
- Clauses not matching any classes in the class model
- Mapping clauses that do not match any objects
- Classes that are not mapped to any component

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.

5. DETAILED EXAMPLE CASES OF ARCHITECTURAL ABSTRACTION EVOLUTION

We evaluated our approach for five open source projects in Section 6. In this section we discuss two of the five projects, Apache CXF [1] and Frag [40], as examples to illustrate our approach in detail. During the incremental refinement of our DSL design we started with examples from these projects and extended the set of samples step-by-step to all changes observed in multiple version of the five cases studied in the next section. At the end of this section we discuss the findings in these examples, while lessons learned are discussed in the next section.

In the Apache CXF example we first created a high level abstraction for CXF 2.0.10 (following the architecture overview shown in Figure 5). Next, we updated it to reflect the changes made in the newer versions of CXF. For Frag we followed the same procedure. 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.

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

Table 1: Architectural abstraction DSL clauses

Version	Files added	Files removed	Files changed	Total changes	DSL changes
CXF Architecture Overview 2.0.10 ⇒ 2.2.12	299	83	1040	1422	1 minor change
CXF Architecture Overview 2.2.12 ⇒ 2.3.7	133	19	923	1075	no changes
CXF Architecture Overview 2.3.7 ⇒ 2.4.3	115	62	739	916	no changes
CXF Transport 2.0.10 ⇒ 2.2.12	29	4	90	123	1 new component
CXF Transport 2.2.12 ⇒ 2.3.7	17	3	117	137	1 new component
CXF Transport 2.3.7 ⇒ 2.4.3	20	23	120	163	1 component removed
Frag 0.6 ⇒ 0.7	48	44	32	124	3 new components, 2 minor changes
Frag 0.7 ⇒ 0.8	9	1	148	158	1 new components, 6 minor changes
Frag 0.8 ⇒ 0.91	7	40	36	83	no changes

Table 2: Necessary changes to DSL code compared to source changes in two examples (Apache CXF, Frag)

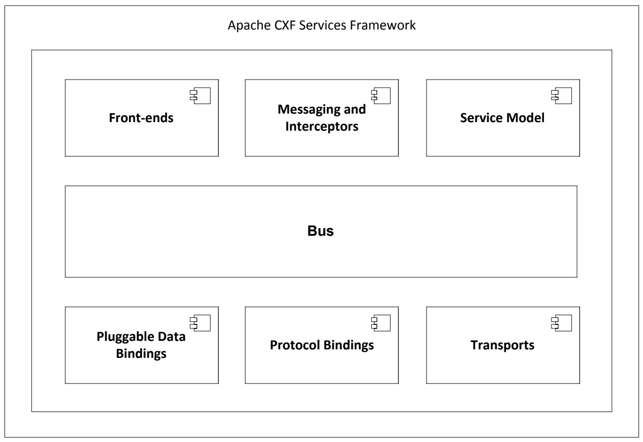

Figure 5: Apache CXF architecture overview [1]

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 one change to the architectural abstraction specification was necessary. This modification constitutes a package introduced between versions 2.2.12 and 2.3.7. This result is based on 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, 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. Since similar architectural abstraction specification changes are shown for other examples, we do not provide a figure listing the changes for this example. The goal of this example was to provide a different (detailed) view for Apache CXF. This view is depicted in Figure 6 and gives an overview of the main transports components.

The high-level architecture of Frag in version 0.91 was shown already in Figure 3. It contains a number of differences when compared to the architecture of version 0.6, which is missing the components DSL, FCL, FMF, and TemplateEngine. The architectural abstraction specification for

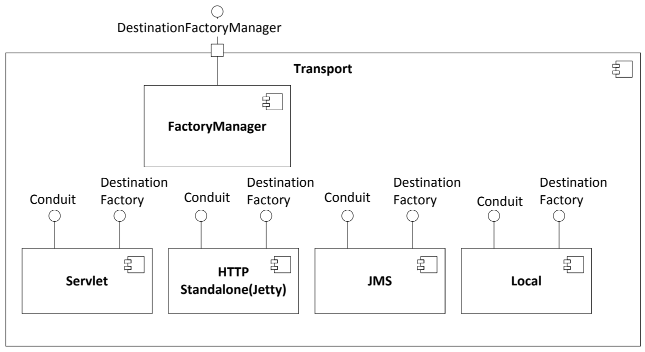

Figure 6: Detail view for Apache CXF transports

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

Figure 7: Architectural abstraction specification modifications for the changes in Frag 0.7

Frag 0.6 has less than fifty lines of code. The changes necessary to conform to Frag 0.7 are shown in Figure 7. 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. This components were grouped in a new package called `mdsd`.

In the following revision of Frag (0.8) only smaller changes to the architecture were made. 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 in the components that are implemented in this package. We used manual search/replace in order to exchange the occurrences of this package. The integration of partial support for automatic architectural abstraction specification updates are a topic for future research. Another change is 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 updates to the architectural abstraction specification are required.

The two cases, Apache CXF and Frag, are described in detail in this section to illustrate our approach. Frag was chosen because of the fact that it shows the modifications caused by major architecture changes. Apache CXF was chosen because each of the revision contains about a thousand changes, but the vast majority of them have no impact on the architecture. Hence, only a very few changes to the architectural abstraction specification are needed in the Apache CXF case.

These two examples confirm that it is possible to create abstractions based on generic filters. They indicate that this 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 is not affected. A `Package` rule that uses a regular expression like ".*model.*" only is affected if this exact part is modified, while a `Package`-rule based on the fully qualified name 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` and the union of their results (`or`). One example is the Shell component in Figure 4 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: They are hard to read because it is unclear which classes match the specified filter. Relationship-based filters can also have side-effects. A related class can reside within a package that is also targeted by a package-based or a name-based filter. However this can be avoided by defining an exception in one of the filters.

The evolution of relationship-based filters also works like the already mentioned `Package`-filter 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.

While we demonstrated our approach in this paper on case studies 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. This limitation often increases the number of rules necessary to define an architectural abstraction specification, though.

In our future work we will evaluate different options for extending our DSL 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.

6. LESSONS LEARNED AND EVALUATION

To validate our approach, we realized architectural abstraction specifications for five existing open source projects [22, 24, 1, 40, 38] of varying size (see Table 3). Samples from two of these evaluations have been discussed in the previous section to illustrate our DSL and its incremental design.

The time needed for creating a new architectural abstraction specification depends on prior existing knowledge of the architecture and source code of the project. Once this knowledge was acquired (obtained from the existing documentation and from studying the source code), creating an architectural abstraction specification took less than 15 minutes for any one of the examples.

Our approach has limitations when 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). Another way to overcome this limitation for existing source code could be to first create an initial architectural view using an automated clustering approach [10] which can later be refined.

Once the architectural abstraction specification is defined we are able to automatically create component models from source code. We noticed that most of our component definitions are based on packages which are the only possibility of grouping multiple Java classes (besides Tagging interfaces and so on). The advantage of component definitions based on packages and existing other groupings 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. E.g. 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.

Whenever a component model is created by the transformation it stores the classes that realize the component in a Realization-Relation as defined by the UML-Standard. This makes tracing the classes responsible for realizing specific components, and vice versa, straightforward.

Our approach supports the creation of architectural abstraction specifications on different levels of abstraction. The data in Table 3 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 CXF to components. This indicates that the CXF architectural abstraction specification is on a higher abstraction level.

For approaches like this, performance often is a big problem. This problem is based on the exponential growth of the execution time 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 3 shows the execution times for five open source cases which we obtained on a 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, we only report the averages here.

Project	#Clauses	LoC	Avg. Exec. Time (in ms)	σ (in ms)
Cobertura	19	85k	1516	368
Frag	41	13k	3559	563
FreeCol	21	103k	9244	1698
Apache CXF	35	386k	19380	1501
Hibernate	96	347k	64626	3413

Table 3: Execution times, standard deviation (σ) and other key data for implemented examples

The results from Table 3 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. Please note that we did not measure the time that is needed for extracting the class model from the source code, since this algorithm only converts every class in the source code into an instance in the model.

7. RELATED WORK

In this section we compare to related approaches that either use similar techniques or try to solve similar problems. We have split the related work into different groups: In Subsection 7.1 we present a number of selected articles that apply different approaches that make use of automatic clustering in one or the other form. Subsection 7.2 discusses articles that propose different kinds of model-based approaches that create abstractions or views from source code. Finally, Subsection 7.3 presents works that focus on model evolution and consistency checking of models.

7.1 Automatic clustering approaches

Abreu et al. introduce a reengineering approach using cluster analysis [2]. It 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 [39]. 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. [7] 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 [5]. As already mentioned in Section 1, the case study by Corazza et al. [7] states that the authoritativness values are close to 0.5 in 5 of 7 cases. This means that in 5 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 [25] 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 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. [10] 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. [23] integrate a latent semantic indexing approach [9] 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. Differential to all automatic clustering approaches, our approach does require manual interaction for the creation and maintenance of the architectural abstraction specification but not after each execution of the transformation.

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 [26]. 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. 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. Scaniello et al. [35] propose an approach for semi-automatically detecting layers in software systems based on the algorithm introduced by J. M. Kleinberg [19]. The authors implemented a prototype and provide a case study for JHotDraw. While their approach is focused 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 that help the understanding of the software system.

Sartipi describes a pattern-based approach for recovering software architecture [34]. It models the process as a graph pattern matching problem between an entity relationship graph and an architecture pattern graph. While this approach uses the two models as input, we use the source code and the architectural abstraction specification in the DSL as input and the resulting component models are only used for consistency checks.

Ivkovic and Kontogiannis [17] provide an approach for keeping models synchronised. 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 [13] 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 for other models but easier to define and allow the definition of architectural abstraction specifications on different levels of abstraction.

Brosig et al. [6] 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 analysing the existing source code.

Another approach for mapping source code models to high-level models in introduced by Murphy et al. [29]. 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.

Mens et al. [28] 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. We plan to further investigate this topic for other architecture documentation such as patterns.

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. [33] 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 [30] with "Constraint with Action Language" [3]. It uses annotated directed acyclic graphs as model representations. This approach has the advantage that is does not work with model comparison. It works directly on the single model and its modifications while our approach works by creating a series of models over time. However, our approach is targeted at creating an abstraction in the form of a component model from source code and keeping track of the model changes is done implicitly by only comparing the different versions of the abstract model.

Passos et al. [32] give a illustrative overview on static architecture-conformance checking. They compare three approaches: The Lattix Dependency Manager (LDM) which is based on Dependency-Structure Matrices, .QL which is a source code query languages (SCQL), and the reflexion models (RM) introduced by Murphy et al. [29]. 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. [14] 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. While their approach has similarities with our approach, their approach offers only limited ways to describe mappings between components and source bode. 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.

8. CONCLUSION

In this paper we presented a semi-automatic approach for supporting architectural abstractions of source code into architectural component models. Based on a DSL, it automatically generates component models 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 create different, abstract views for one project. Another 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 current evaluations and experience show that architectural abstraction specifications can be created and maintained with relatively low effort. We can test for inconsistencies between abstraction and code and raise an error if 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.

9. REFERENCES

[1] Apache CXF. `http://cxf.apache.org`, 2011.

[2] F. B. e. Abreu, G. Pereira, and 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, pages 13–, Washington, DC, USA, 2000. IEEE Computer Society.

[3] S. A. Ajila and S. Alam. Using a formal language constructs for software model evolution. In *Proceedings of the 2009 IEEE International Conference on Semantic Computing*, ICSC '09, pages 390–395, Washington, DC, USA, 2009. IEEE Computer Society.

[4] S. Beydeda and V. Gruhn. *Model-Driven Software Development.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[5] R. A. Bittencourt and 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*, pages 251–254, Washington, DC, USA, 2009. IEEE Computer Society.

[6] F. Brosig, S. Kounev, and 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, pages 10:1–10, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[7] A. Corazza, S. Di Martino, and 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, pages 88–96, Washington, DC, USA, 2010. IEEE Computer Society.

[8] R. M. Davison, M. G. Martinsons, and N. Kock. Principles of canonical action research. *Information Systems Journal*, 14:65–86, 2004.

[9] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41:391–407, 1990.

[10] J. Dietrich, V. Yakovlev, C. McCartin, G. Jenson, and M. Duchrow. Cluster analysis of Java dependency graphs. In *Proceedings of the 4th ACM symposium on Software visualization*, SoftVis '08, pages 91–94, New York, NY, USA, 2008. ACM.

[11] Eclipse Foundation. EMF Compare. `http://www.eclipse.org/emf/compare/`, 2011.

[12] Eclipse Foundation. Xtext. `http://www.eclipse.org/Xtext/`, 2011.

[13] 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, volume 2984 of *Lecture Notes in Computer Science*, pages 37–53. Springer, 2004.

[14] M. Feilkas, D. Ratiu, and E. Jurgens. The loss of architectural knowledge during system evolution: An industrial case study. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 188 –197, 2009.

[15] M. Fowler. *Domain-Specific Languages (Addison-Wesley Signature Series (Fowler))*. Addison-Wesley Professional, 1 edition, 2010.

[16] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 1 edition, 2009.

[17] I. Ivkovic and K. Kontogiannis. Tracing evolution changes of software artifacts through model synchronization. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 252–261, Washington, DC, USA, 2004. IEEE Computer Society.

[18] A. Jansen, J. van der Ven, P. Avgeriou, and D. K. Hammer. Tool support for architectural decisions. In *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture*, WICSA '07, pages 4–, Washington, DC, USA, 2007. IEEE Computer Society.

[19] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46:604–632, 1999.

[20] N. Kock. The three threats of action research: a discussion of methodological antidotes in the context of an information systems study. *Decis. Support Syst.*, 37:265–286, 2004.

[21] M. Kofman and E. Perjons. Metadiff - a model comparison framework. `metadiff.sourceforge.net/docs/metadiff.pdf`.

[22] J. Linwood and D. Minter. *Beginning Hibernate, Second Edition*. Apress, Berkely, CA, USA, 2nd edition, 2010.

[23] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.*, 16, 2007.

[24] M. Doliner, J. Erdfelt, J. Lewis, G. Lukasik, J. Mareš, and J. Thomerson. Cobertura. `http://cobertura.sourceforge.net`, 2011.

[25] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 103–112, Washington, DC, USA, 2001. IEEE Computer Society.

[26] O. Maqbool and H. Babri. Hierarchical clustering for software architecture recovery. *IEEE Trans. Softw. Eng.*, 33:759–780, 2007.

[27] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26:70–93, 2000.

[28] K. Mens, T. Mens, and M. Wermelinger. Maintaining software through intentional source-code views. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, SEKE '02, pages 289–296, New York, NY, USA, 2002. ACM.

[29] G. C. Murphy, D. Notkin, and 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, pages 18–28, New York, NY, USA, 1995. ACM.

[30] Object Management Group. OCL 2.2 Specification, 2010.

[31] Object Management Group. UML 2.3 Superstructure, 2010.

[32] L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. das Chagas Mendonca. Static architecture-conformance checking: An illustrative overview. *IEEE Softw.*, 27:82–89, 2010.

[33] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook, and M. Chechik. Consistency checking of conceptual models via model merging. In *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, pages 221 –230, 2007.

[34] K. Sartipi. Software architecture recovery based on pattern matching. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 293–, Washington, DC, USA, 2003. IEEE Computer Society.

[35] G. Scanniello, A. D'Amico, C. D'Amico, and T. D'Amico. An approach for architectural layer recovery. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2198–2202, New York, NY, USA, 2010. ACM.

[36] Soyatec. eUML2. `http://www.soyatec.com/euml2/`, 2011.

[37] D. Spinellis. UMLGraph. `http://www.umlgraph.org`, 2011.

[38] The Freecol Team. Freecol. `http://freecol.org`, 2011.

[39] M. von Detten and 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, pages 23–32, New York, NY, USA, 2011. ACM.

[40] U. Zdun. The Frag Language. `http://frag.sourceforge.net/`, 2011.

[41] O. Zimmermann, U. Zdun, T. Gschwind, and F. Leymann. Combining pattern languages and architectural decision models in a comprehensive and comprehensible design method. In *Working IEEE/IFIP Conference on Software Architecture (WICSA) 2008*, Vancouver, BC, Canada, 2008.