Architecting for decision making about code evolution

Thomas Haitzer
Software Architecture Group
University of Vienna
Austria
thomas.haitzer@univie.ac.at

Elena Navarro
Department of Computing Systems
University of Castilla-La Mancha
Spain
elena.navarro@uclm.es

Uwe Zdun
Software Architecture Group
University of Vienna
Austria
uwe.zdun@univie.ac.at

ABSTRACT

During software evolution, it is important to evolve not only the source code, but also its architecture to prevent architecture drift and architecture erosion. This is a complex activity, especially for large software projects, with multiple development teams that might be located in different countries or on different continents. To ease this kind of evolution, we have developed a domain-specific language for making decisions about the evolution. It supports the definition of architectural changes based on multiple implementation tasks that can have temporal dependencies among each other. Then, by means of a model-to-model transformation, we automatically create a constraint model that we use to generate, by means of the Alloy model analyzer, the possible alternative decisions for executing the implementation tasks. The tight integration with architecture abstractions enables architects to automatically check the changes related to an implementation task in relation to the architecture description. This helps keeping architecture and code in sync, avoiding drift and erosion.

Categories and Subject Descriptors

D.2.11 [Software Engineering]: Software Architectures;

D.2.2 [Software Engineering]: Design Tools and Techniques

Keywords

Architecture Evolution, Evolution Planning, Design Rationale, Architecture Documentation

1. INTRODUCTION

Software Evolution has always been and still is one of the challenging activities of the software lifecycle. Basically, from the first steps of a software project onwards, the need of change starts to arise because new market needs constantly impose new requirements, supporting technology is updated, decisions about the software system change, and so on. In this context, the use of Software Architecture (SA) has been highlighted as an important asset because, SA can be used as an artifact for the evolution to guide the planning and restructuring of the software [7,14], but it is also an artifact of the evolution, because it must be evolved itself [2]. Moreover, SA evolution is a complex activity, especially for largo software projects with multiple development teams that might be located in different countries that work on different parts of the project in parallel, so there is a clear need to manage properly who is in charge of each requested change and how and when it will be carried out.

ECSAW '15, September 07 - 11, 2015, Dubrovnik/Cavtat, Croatia

DOI: http://dx.doi.org/10.1145/2797433.2797487

To support stakeholders with methods and tools to help in the evolution process, we have developed a DSL for making decisions about the evolution that provides architects with expressive power to describe which implementation tasks must be performed by the development team and which temporal dependencies among these tasks exist. Once the architect has specified these implementation tasks with the DSL, by using a model-to-text transformation that we have implemented in Xtext [8], they are translated to Alloy [16] to evaluate which are the possible decisions for realizing the architecture evolution in terms of the specified implementation tasks. We have integrated this approach with our architecture abstraction specification language [13], so that architects and/or developers can automatically update the specification of the architecture which helps keeping two important assets of a software project (the SA and the source code) in sync. This way, our approach does not only allow to evolve SA and source code in sync, but also requires to define the architectural changes only once (when defining the implementation task) and the architecture description is updated automatically. Furthermore, we can use the Architecture Abstraction DSL's benefits which we described in our previous work [13]. This proposal provides several advantages:

- First, it provides the software architect with facilities to automatically generate decision alternatives for carrying out the implementation tasks so that they can be easily distributed among the teams or team members.
- Second, it releases the software architect from the burden to manually update the architectural description because the defined implementation tasks are used to *automatically update the architecture specification*. This is very important as it helps to avoid the architecture drift and architecture erosion that usually emerge during the evolution.
- Third, the defined implementation tasks serve for the purpose of *creating a documentation of the evolution*. This is a very important question as several studies [5][21] carried out with subjects from both industry and academia have concluded that using the architectural documentation the time necessary to carry out the change-tasks could be shorten.

This paper is structured as follows. After this introduction, related works are analyzed and compared to our proposal. Then, in Section 3, we first present the DSL we have developed, and then

we discuss the support for deciding on implementation steps. A case study that illustrates the feasibility of the approach is shown in Section 4. Finally, Section 5 concludes this work.

2. RELATED WORK

A number of approaches have been proposed that focus on the evolution of a software system. While some of these approaches focus on transforming a legacy system's architecture [6,15], others focus on the concept of Evolution styles [11,20,24], which are specifically defined for a set of architectural styles and constrain the evolution of the system. Cuesta et al. [7] extend this notion and propose the use of architectural knowledge-driven evolution styles to document a system's evolution.

However, these approaches focus on capturing the knowledge about the architecture evolution, while our approach, which also captures the knowledge in the form of described (documented) tasks that have been executed, has as its main goal to ease the complexity inherent to architecture evolution and help on deciding on how to execute the defined implementation tasks.

Garlan et al. [10] propose a tool called Ævol, that supports the definition and planning of architecture evolution based on Evolution styles. It allows the specification of evolution paths and the comparison of alternative evolution paths regarding correctness conditions and cost-benefit. While their approach has a similar focus, it requires the architect to specify all possible evolution paths, whereas our approach only requires the definition of the necessary evolution tasks and their dependencies, being generated automatically all possible decisions for executing the tasks.

Barnes et al. [3] extend Ævol and use existing automatic planning tools for the generation of possible evolution paths. While their approach is similar to our approach, as it finds possible evolution paths to evolve the current architecture to a target architecture, our approach differs substantially in other aspects. While our approach provides a user-friendly DSL for defining tasks that are automatically translated into an Alloy model, they need to manually represent their evolution steps in the planning domain definition language (PDDL). Furthermore, the tight integration of architectural descriptions and implementation tasks in our approach makes it possible to apply changes to the architecture description that are specified in the tasks automatically once the tasks are completed.

Ajila and Alam [1] have proposed a formal language based on the Object Constraint Language to construct evolution models. Based on this language, they provide support for automatic dependency analysis of the model. Our approach, however, focuses on planning the tasks that need to be completed during the evolution, while the evolution of the architectural description is not the sole purpose of our approach.

McVeigh et al. [18] proposed Evolve, a model driven tool that captures incremental change in the definition of software architecture. It implements Backbone, an architectural description language with a graphical (UML2) and a textual representation. Based on the architecture documentation, backbone directly constructs initial implementations and extensions to these implementations. While their approach supports the evolution of a system's architecture, our approach focuses on managing the complexity inherent to evolution and provides valid plans for executing the tasks that arise during the evolution of a system.

An approach called ADVERT is proposed by Konersmann et al. [17]. This approach provides support for evolution of an architectural level by maintaining tracelinks between requirements, design decisions, and architectural elements and also including software architecture information into the source code. However, their approach does not consider planning the evolution and thus has a focus different from our approach. In addition, their approach was only partially implemented, being other elements assumed to work and only described for EJBs. However, our approach is programming language independent.

Grunske [12] proposes an approach for architectural refactoring based on a hypergraph-based data structure that allows the formalization of refactorings as hypergraph transformation rules that can be applied automatically. However, unlike our approach, this approach does not support evolution in general but is limited to behavior preserving architectural refactoring.

A number of different approaches focus on change impact analysis and utilize models and model conformance checking, reasoning on ontologies [4,22], or Bayesian Belief Networks [25] to analyze the impact of changes on systems properties. Others propose the use of anti-patterns to detect conflicting Architectural Design Decisions during evolution [19]. But all these approaches focus on analyzing the changes to a system during the evolution but none of them focuses on deciding the evolution itself.

Summarizing, most of the approaches discussed in this section focus on the documentation of the evolution, while our approach focuses on decisions about the evolution. The approaches that focus on automatic evolution either lack the support for automatically changing the architecture description, which requires the architect to perform this task manually, or are focused on specific subsets of systems.

3. ARCHITECTING FOR DECIDING CODE EVOLUTION

Whenever the code is being developed, the coding tasks are usually carried out in an iterative manner, so that no new component is developed from its very beginning to its end, but usually different components can be developed in parallel. However, the main problem is that there are, usually, *internal dependencies* among them that must be identified and considered whenever a system is being developed. These internal dependencies impose mainly *temporal constraints,* in terms of when the different features supported by each component should be developed. Let us illustrate this problem with a scenario, which we will use as a running example in the remainder of this paper. As shown in Figure 1(a), initially two components *ComponentA* and *ComponentB* communicate with each other directly through a connector. Now let us assume that, due to new requirements, a distribution of these two components on different servers is necessary. This leads to an Architectural Design Decision (ADD) to implement a version of the broker pattern between these two components. This architectural change is shown in Figure 1(b).

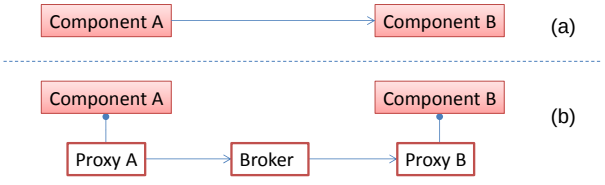

Figure 1: Architecture changes in the Broker scenario

The broker pattern is a pattern for communication between

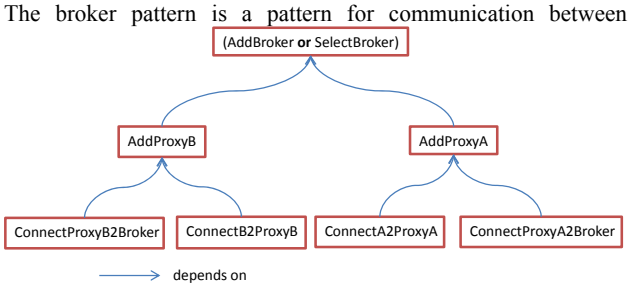

Figure 2: Planning an Evolution Step

distributed objects. This ADD leads to a number of *design decisions* and thus a number of different *implementation tasks*, whose timing is constrained by internal dependencies, as shown Figure 2. Specifically, the following tasks need to be implemented in order to complete the implementation of this ADD.

- For using the Broker itself, a suitable middleware framework must be set up and configured.
- The above mentioned proxies for the two components need to be created.
- The proxies need to be wired to the broker. Moreover, the components need to be changed in order facilitate the new communication form. The direct connector needs to be removed and the usage of the proxies needs to be implemented. If dependency injection (DI) is used, at least the DI configuration needs to be changed, even if no changes to the components' implementation are necessary.

All of these tasks need to be completed in order to fully comply with the ADD to implement the broker pattern. Even in this small example, a number of temporal dependencies exist between the tasks at hand. The implementation of the proxies requires that the middleware for the Broker is set up and configured, the changes to *Component A* require the existence of the proxy for *Component A*, and the wiring of *Component A* with its proxy requires that the changes to the *Component A* itself are completed. The same or at least similar dependencies exist for *Component B*. These dependencies impose some order in which these tasks need to be completed. In a real world scenario with multiple development teams and more than two components involved in an architectural decision, this problem's complexity grows much further.

In our approach, a software architect defines the tasks and the constraints on the timing of the tasks (e.g. proxy must be implemented before proxy can be connected and used) in a domain specific language especially designed for planning code evolution, called *Evolution DSL*. This DSL allows the architect to specify: (i) a textual description of the implementations task, including any references to relevant ADDs; (ii) the temporal constraints or dependencies of the task; (iii) as well as the changes to the system's architectural description based on the Architecture Abstraction DSL. We describe the technical details and the Evolution DSL in Section 3.1.

Based on these task definitions, our approach supports the architect and the developers during the evolution by automating the complex task of creating the possible decision alternatives for executing the given implementation tasks. We utilize the Alloy[1] model finder for automatically providing multiple possible alternatives for the order of the implementation tasks. These models are provided in graphical and textual form by Alloy. While the textual form supports an automatic interpretation, the graphical form shows which tasks do not have any dependencies to other tasks and thus can be implemented in a parallel fashion without running into any dependency issues. It also supports *easy identification of crucial tasks that need to be completed early*, as well as sets of implementation tasks that do not have dependencies outside the given set.

In our running example, two such sets can be identified: The first contains all tasks related to *Component A* and the second contains all tasks related to *Component B*, while the set up and configuration of the middleware of the Broker qualifies as a crucial task that might hinder further work as both identified sets depend on this task. The sets around *Component A* and *Component B* are good candidates for being developed by the same development team, because this team then can work independently from the other team(s) and is not hindered by any dependencies to tasks that are implemented elsewhere once the set up and configuration of the middleware of the Broker component is completed. Furthermore, the automatically generated decision alternatives ensure that no implementation tasks are started, before their dependencies are fulfilled. Finally, the *defined implementation tasks are used to automatically update the architecture description and serve for the purpose of creating a documentation of the evolution*. The technical details of this support are provided in section 3.2.

3.1 DSL for specifying the code evolution

In this section we describe the concepts and implementation of our Evolution DSL in detail. An important feature of the Evolution DSL is the tight integration with the architecture description itself, which enables to automatically apply the changes, specified in an implementation task to the architecture description, once it is completed. This *releases the software architect from the burden to manually update the architectural description after an implementation task is completed.*

This is why we have integrated the Evolution DSL, which was implemented in Xtext [8], with the Architecture Abstraction DSL that we developed in a previous work [13]. For space reasons, we introduce just briefly our Evolution DSL[2] in this paper.

In order to facilitate the understanding of this paper, Figure 3 shows an excerpt of the grammar for the definition of implementation tasks and the temporal rules for implementation tasks as well as the architectural changes supported. In the rule definition *AddComponentTask*, we can see that the Architecture Abstraction DSL's rules are reused. This enables the automatic application of the architectural changes from completed implementation tasks to the architectural description. This has been implemented as an Eclipse wizard that enables the architect to select which implementation tasks have been already completed and then, using a model-to-model transformation, to update the architecture description.

---

[1] Alloy [16] is a language to formally describe structures and a solver that takes the constraints of a model and finds structures that satisfy them.

[2] For a complete specification & source code see: https://swa.univie.ac.at/DSL_for_planning_the_evolution

```
ImplementationTask:
'Task' name=ID ':'
('status:' status=STATUS)?
('description:' description=STRING)?

// temporal rules
('precedes' precedes+=[ImplementationTask] (',' precedes+=[ImplementationTask])*)?
('directly precedes' directlyprecedes+=[ImplementationTask] (',' directlyprecedes+=[ImplementationTask])*)?
('henceforth requires' requires+=LogicRule (',' requires+=LogicRule)*)?
('in parallel with' inParallelWith+=[ImplementationTask] (',' inParallelWith+=[ImplementationTask])*)?
('succeeds' succeeds+=[ImplementationTask] (',' succeeds+=[ImplementationTask])*)?
('directly succeeds' directlysucceeds+=[ImplementationTask] (',' directlysucceeds+=[ImplementationTask])*)?
(optional?='is optional')?
('is incompatible with' prevents+=[ImplementationTask] (',' prevents+=[ImplementationTask])*)?
architectureChange=ArchitectureChange;

ArchitectureChange:
AddFeatureTask | AddConnectorTask | RemoveFeatureTask | RemoveConnectorTask | AddComponentTask |
RemoveComponentTask | ModifyComponentTask | ComplexTask;

ComplexTask:
'consists of:'
tasks+=TaskReference (',' tasks+=TaskReference)*;

AddComponentTask:
'architecture changes:'
'add component to' transformation=[archDSL::Transformation|TASKS_QUALIFIED_NAME]
componentToAdd=ComponentDef;
```

Figure 3: Excerpt of the Xtext grammar for the Evolution DSL showing the rule for an impl. task, the different types of tasks and two of the rules for specific tasks

An example of different tasks expressed in the DSL is presented in Figure 4. In this example, the complex task of adding a Broker between two Components *A* and *B* is divided into multiple subtasks, which consist of implementing the Broker itself (*AddBrokerFeature*), implementing the Proxies for Components *A* and *B* (*AddProxyA*, *AddProxyB*) and wiring all the components together. Some of these tasks have (temporal) dependencies. In this example, *ConnectA2Proxy* requires that *ProxyA* has been implemented before Component *A* can be wired to *ProxyA*. Also, *ConnectA2Proxy* is itself a complex task that consists of two subtasks, which should be carried out in close succession. In Figure 4 we skipped the tasks regarding Component *B* as they are very similar to the tasks regarding Component *A*.

Other dependencies that stem from organizational requirements (e.g. that the tasks will be split between independent teams of developers) can be modelled in the same way as constraints resulting from implementation itself.

3.2 Generating decision alternatives for evolution

Once the tasks are defined, we use the features provided by Xtext to automatically execute a model-to-text transformation that creates an Alloy model, which is used to generate the possible decision alternatives. Alloy [16] is a structural modelling language based on first-order logic for expressing complex structural constraints and behavior. The Alloy Analyzer is a constraint solver that provides fully automatic simulation and checking. It allows us to define the concepts of basic and complex implementation tasks, the definition of specific implementation tasks and their constraints based on the abstract concepts, as well as the following (summarized) constraints that need to hold for all implementation task models:

- An implementation task is followed by a set of implementation tasks (*next* relation).
- A complex implementation task is an implementation task that consists of a set of implementation tasks (*consistsOf* relation).
- All defined implementation tasks need to be acyclic with respect to the *next* relation as well as the *consistsOf* relation.
- All defined implementation tasks need to exist in the solution and must be reachable. Either they are part of the initial tasks or they are reachable through an initial task.
- A complex implementation task is immediately followed by one of its subtasks.
- A complex implementation tasks precedes all its subtasks.
- Each implementation task can only be part of zero or one complex implementation tasks.

We show an excerpt of the Alloy code that was generated for the Broker example in Figure 5. In particular, we show the constraints that ensure that: (i) a task only occurs once, (ii) a task cannot be part of itself, (iii) the definition of the tasks *AddProxyA* and

```
Task AddBroker:
description: "Tasks necessary for adding
the new broker to the architecture"
consists of:
AddBrokerFeature,
AddProxyA,
AddProxyB,
ConnectA2Proxy,
ConnectProxyA2Broker,
ConnectB2ProxyB,
ConnectProxyB2Broker

Task AddBrokerFeature:
description: "implement the broker functionality"
architecture changes:
add component to Frag
Component Broker consists of
Package("univie.swa.example.broker")

Task UpdateComponentA:
directly precedes ConnectA2Proxy
architecture changes:
replace feature Frag.ComponentA :
Package_univie_swa_example_original_package
with new feature:
Package("univieswa.example.A.usingBroker")
after Frag.ComponentA.Package_univie_swa_example_original_package
```

```
Task ConnectA2ProxyA:
description: "implement the conn. between comp. A and proxy A"
precedes ConnectB2ProxyB
succeeds AddProxyA
architecture changes:
add connector to Frag.ComponentA
connector to AddProxyA.ProxyA

Task ConnectProxyA2Broker:
description: "wire the proxy and the broker together"
succeeds AddProxyA,AddBrokerFeature
architecture changes:
add connector to AddProxyA.ProxyA
connector to AddBrokerFeature.Broker

Task AddProxyA:
description:"implement the proxy that hides the broker from comp. a"
architecture changes:
add component to Frag
Component ProxyA consists of
Package("univie.swa.example.proxyA")

Task ConnectA2Proxy:
succeeds AddProxyA
consists of:
UpdateComponentA,
ConnectA2ProxyA
```

Figure 4: Excerpt from the implementation tasks of the example for adding a Broker.

```
//…
fact AcyclicImplementationTasks {
   no task: ImplementationTask| task in task.^next
}
fact AcyclicComplexImplementationTasks {
   no task: ComplexImplementationTask | task in task.^consistsOf
}
// …
one sig AddProxyA extends ImplementationTask {}
one sig ConnectA2ProxyA extends ImplementationTask {}
// …
pred show {
//..
all s1: AddProxyA, s2: ConnectA2ProxyA | s2 in s1.^next
}
run show for 5
```

Figure 5: Excerpt of the Alloy code for the introduce Broker example

ConnectA2ProxyA as implementation tasks, (iv) finally, *AddProxyA* needs to be executed before the task *ConnectA2ProxyA*.

We then use the Alloy tool (version 4.2) to create multiple possible decision models that adhere to the identified constraints. These models are provided in a textual and a graphical representation by the tool. Figure 6 shows a possible order of the implementation tasks for the Broker example generated by Alloy.

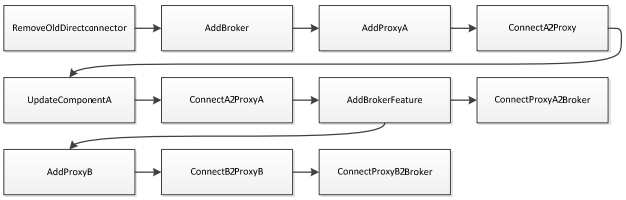

Figure 6: Decision alternative generated by Alloy for the Broker example

Please note that a limitation of this approach arises through the use of Alloy, which, as a model finder that uses SAT solving for finding model instances, requires a suitable scope, as within this scope, the search for a model is complete, while the search itself is incomplete. For all our models, we chose a default scope of 5, because it is enough to find multiple solutions for all our generated models. Due to the size of architectural component models and due to the fact that our Alloy models do not have free variables, our experience shows that for this subset of models a model instance can be found. If no model instance is found, the bound can be raised.

It is worth noting that this approach has been designed for evolving architecture and code in sync. When the code is changed first, the features of the Abstraction DSL can aid in ensuring consistency between architecture and code.

4. CASE STUDY

In this section we describe our case study of Soomla, an open source framework for virtual economy operations in a single, cross-platform, SDK mainly used for mobile games [23]. In our case study, we describe the changes that were implemented from Version 3.2 to Version 3.3. Figure 7 shows an overview of Soomla's architecture and the respective changes to the architecture. In Version 3.2 Soomla's billing system only integrated the Billing API for Android provided by Google which was directly used throughout the system. However, since the need arose to support other billing providers as well, this was no longer suitable and the system needed to be evolved.

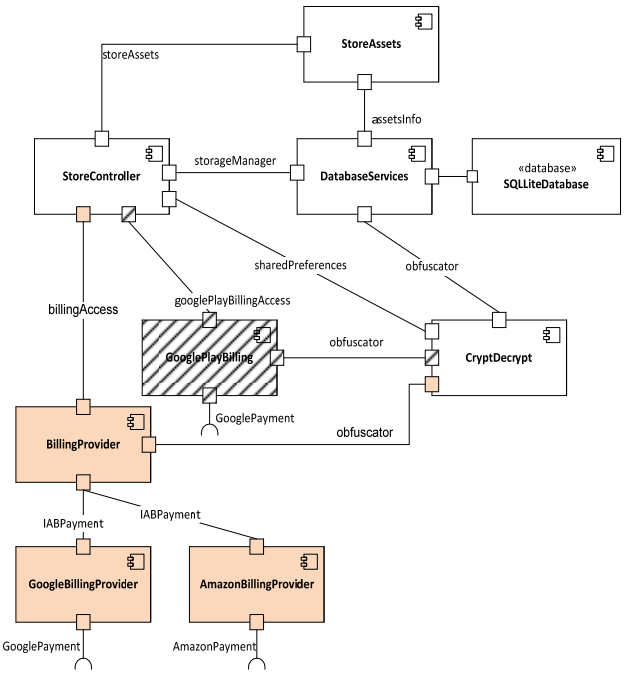

Figure 7: Architecture overview of Soomla with changes between version 3.2 and version 3.3

We described this evolution as a set of implementation tasks which replace the original provider-dependent *GooglePlayBilling* component with a new provider-independent billing component, and then (re-)implement the provider-specific parts based on the new billing infrastructure. The detailed implementation tasks and their architectural changes are shown in Figure 8.

Based on our description of the implementation tasks, an Alloy model was automatically generated by our model-to-text transformation implemented in Xtend. We then used the Alloy model finder to create the decision alternatives for executing the implementation tasks without violating any constraints. This was computed by Alloy in 149 ms and resulted in multiple possible alternatives for executing the implementation tasks at hand. This order ensures that all constraints are satisfied throughout the execution of the different implementation tasks

Once the implementation tasks were completed, we automatically added the architectural changes from the implementation tasks to the architectural description of Soomla using our wizard (see Figure 9), which we integrated into the DSL user-interface. This wizard then uses Xtend [9] to apply the changes to the architecture description written in the Architecture Abstraction DSL.

This case study, as well as the running broker example, shows the applicability of the approach with respect to feasibility. The time required for finding suitable plans with Alloy was around 150ms for all presented examples on a Lenovo Thinkpad X240 with i5 Processor and 8 Gb RAM and a Samsung Evo 840 SSD. We think that in large projects with multiple developer teams, the effort necessary to use our approach is outweighed by the benefits of having a plan for executing the given tasks that shows which tasks can be executed in parallel, as well as which tasks are prerequisites to other tasks and thus should be prioritized.

```
Task ProviderIndependentBilling:
consists of:
ImplementBillingComponent,
WireBillingComponent,
SubstituteBillingInStoreController,
ImplementNewGoogleBillingProvider,
RemoveOldGoogleBillingProvider

Task ImplementBillingComponent:
description: "Implement a new abstract billing provider that is independent from any actual billing providers"
architecture changes:
add component to Soomla
Component Billing
consists of      Package("root.com.soomla.store.billing",excludeChildren)

Task WireBillingComponent:
succeeds ImplementBillingComponent
architecture changes:
add connector to ImplementBillingComponent.Billing connector to Soomla.CryptDecrypt

Task SubstituteBillingInStoreController:
succeeds ImplementBillingComponent
consists of:
ConnectToAbstractBilling,
RemoveConntectorGooglePlayBilling

Task ConnectToAbstractBilling:
architecture changes:
add connector to Soomla.StoreController connector to ImplementBillingComponent.Billing

Task RemoveConntectorGooglePlayBilling:
precedes RemoveOldGoogleBillingProvider
architecture changes:
remove connector from Soomla.StoreController : connector_GooglePlayBilling

Task ImplementNewGoogleBillingProvider:
succeeds ImplementBillingComponent
architecture changes:
add component to Soomla
Component GoogleBilling
consists of
{
   Package("root.com.soomla.store.billing.google")
   or {
      Package("root.com.soomla.store.billing.google")
      and
      InstanceOf("root.com.soomla.store.billing.IIabService")
   }
}
connector to ImplementBillingComponent.Billing

Task RemoveOldGoogleBillingProvider:
architecture changes:
remove component Soomla.GooglePlayBilling
```

Figure 8: Implementation tasks for Soomla v3.2 to v3.3

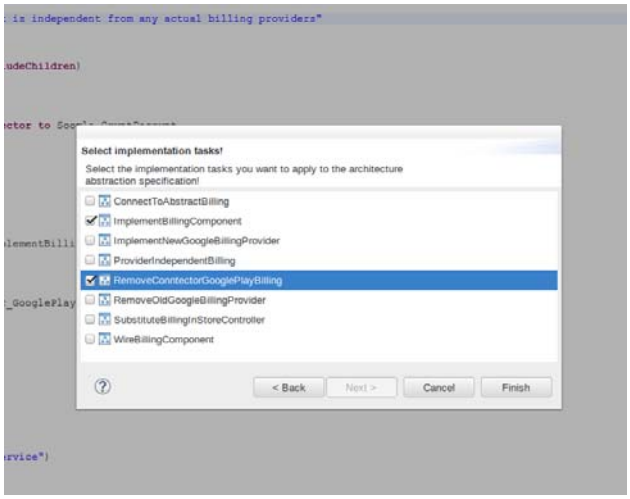

Figure 9: Wizard integrated into the DSL user-interface to add the architectural changes

5. CONCLUSION

In this paper we present an approach for ensuring consistency between two important assets of a software project, namely software architecture and source code, during the evolution of a (large) system by describing an evolution as a set of implementation tasks. We provide a DSL that supports the description of implementation tasks based on their effects on a system's architecture, as well as the (temporal) constraints that exist between different implementation tasks. Besides the value of this DSL for documentation of architecture evolution, our approach supports tool-based guidance throughout the implementation tasks necessary for performing evolution. That is, based on the implementation task descriptions, we use Alloy models to calculate possible decision alternatives for code evolution under the given constraints that ensure the consistency of the evolution or warn the software developer if no viable code evolution decisions can be found. The integration with the architecture description helps keeping software architecture and source code in sync, avoiding drift and erosion. We show the applicability of the approach in a running example based on the implementation of the Broker pattern in an application as well as a real-life scenario for the evolution of the open-source in-app-purchase framework Soomla. In our future work we will perform a case study with developers and architects to better determine the approach's benefits with respect to its costs.

6. ACKNOWLEDGMENTS

This research has been partly funded by the Spanish Ministry of Economy and Competitiveness and by the FEDER funds of the EU under the project Grant insPIre (TIN2012-34003) and by the Ministry of Education, Culture and Sport under the State Program to Promote Talent and Employability in I+D+I, National Sub-Program for Mobility belonging to the Spanish National Plan for Scientific and Technical Research and Innovation 2013-2016 (CAS14/00020).

7. REFERENCES

[1] Ajila, S. a. and Alam, S. Using a Formal Language Constructs for Software Model Evolution. *2009 IEEE International Conference on Semantic Computing*, 390–395. 2009.

[2] Barais, O., Le Meur, A.F., Duchien, L., and Lawall, J. Software Architecture Evolution. In *Software Evolution*, Tom Mens and Serge Demeyer (eds.). Springer Berlin Heidelberg, 233–262. 2008.

[3] Barnes, J.M., Pandey, A., and Garlan, D. Automated planning for software architecture evolution. *28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013)*, IEEE, 213–223. 2013.

[4] De Boer, R.C., Lago, P., Telea, A., and van Vliet, H. Ontology-driven visualization of architectural design decisions. *Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA 2009)*, IEEE, 51–60. 2009.

[5] Bratthall, L., Johansson, E., and Regnell, B. Is a Design Rationale Vital when Predicting Change Impact? – A Controlled Experiment on Software. *2nd International Conference on Product Focused Software Process Improvement (PROFES 2000)*, Springer, 126–139. 2000.

[6] Correia, R., Matos, C., El-Ramly, M., Heckel, R., Koutsoukus, G., and Andrade, L. *Software Engineering at the Architectural Level: Transformation of Legacy Systems*. Department of Computer Science, University of Leicester, UK. 2002.

[7] Cuesta, C.E., Navarro, E., Perry, D.E., and Roda, C. Evolution styles: using architectural knowledge as an evolution driver. *Journal of Software: Evolution and Process* 25, 9, 957–980. 2013.

[8] Eclipse. Xtext. Retrieved April 24, 2015 from https://eclipse.org/Xtext

[9] Eclipse. Xtend. Retrieved April 24, 2015 from https://www.eclipse.org/xtend

[10] Garlan, D., Barnes, J.M., Schmerl, B.R., and Celiku, O. Evolution styles: Foundations and Tool support for Software Architecture Evolution. *Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA 2009)*, IEEE, 131–140. 2009.

[11] Le Goaer, O., Tamzalit, D., Oussalah, M.C., and Seriai, A.-D. Evolution styles to the rescue of architectural evolution knowledge. *3rd International workshop on SHAring and Reusing architectural Knowledge (SHARK'08)*, ACM Press, 31–36. 2008.

[12] Grunske, L. Formalizing architectural refactorings as graph transformation systems. *Proceedings - Sixth Int. Conf. on Softw. Eng., Artificial Intelligence, Netw. and Parallel/Distributed Computing and First ACIS Int. Workshop on Self-Assembling Wireless Netw., SNPD/SAWN 2005*, 324–329. 2005.

[13] Haitzer, T. and Zdun, U. Semi-automated architectural abstraction specifications for supporting software evolution. *Science of Computer Programming* 90, 135–160. 2014.

[14] Holt, R. Sofware Architecture as a Shared Mental Model. *Proceedings of the ASERC Workshop on Software Architecture*. 2002.

[15] Hunold, S., Korch, M., Krellner, B., Rauber, T., Reichel, T., and Rünger, G. Transformation of Legacy Software into Client/Server Applications through Pattern-Based Rearchitecturing. *32nd Annual IEEE International Computer Software and Applications Conference (COMSAC'08)*, IEEE, 303–310. 2008.

[16] Jackson, D. *Software Abstractions. Logic, Language and Abstractions*. MIT Press. 2011.

[17] Konersmann, M., Durdik, Z., Goedicke, M., and Reussner, R.H. Towards Architecture-centric Evolution of Long-living Systems (the ADVERT Approach). *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, 163–168. 2013.

[18] McVeigh, A., Kramer, J., and Magee, J. Evolve: tool support for architecture evolution. *2011 33rd International Conference on Software Engineering (ICSE)*, 1040–1042. 2011.

[19] Navarro, E., Cuesta, C.E., Perry, D.E., and González, P. Antipatterns for Architectural Knowledge Management. *International Journal of Information Technology & Decision Making* 12, 3, 547–589. 2013.

[20] Noppen, J. and Tamzalit, D. ETAK: Tailoring Architectural Evolution by (re-)using Architectural Knowledge. *ICSE Workshop on Sharing and Reusing Architectural Knowledge (SHARK '10)*, ACM Press, 21–28. 2010.

[21] Ozkaya, I., Wallin, P., and Axelsson, J. Architecture knowledge management during system evolution. *2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge (SHARK '10)*, ACM Press, 52–59. 2010.

[22] Pahl, C., Giesecke, S., and Hasselbring, W. Ontology-based modelling of architectural styles. *Information and Software Technology* 51, 12, 1739–1749. 2009.

[23] SOOMLA. Open source framework version 3.1. Retrieved April 24, 2015 from http://soom.la/

[24] Tamzalit, D., Oussalah, M.C., Le Goaer, O., and Seriai, A.-D. Updating software architectures : A style-based approach. *International Conference on Software Engineering Research and Practice (SERP 2006)*, CSREA Press, 313–318. 2006.

[25] Tang, A., Nicholson, A.E., Jin, Y., and Han, J. Using Bayesian belief networks for change impact analysis in architecture design. *Journal of Systems and Software* 80, 1, 127–148. 2007.