A Domain Specific Language for Stepwise Design of Software Architectures

Fabian Gilson and Vincent Englebert
PReCISE Research Center, Faculty of Computer Science, University of Namur, Namur, Belgium
{fabian.gilson, vincent.englebert}@unamur.be

Keywords: Software Architecture, Design Method, Design Rationale, Traceability, Model Transformation.

Abstract: Stakeholders have to face requirements in increasing number and complexity. Their translations to system functionalities are often diluted into the overall architecture so that it becomes tricky to undertake future changes. Since information systems are intended to evolve in terms of functionalities and underlying technologies, the link between requirements and design artifacts is primordial. Agile design methods and documentation techniques have emerged in the past years in order to deal with the amount of requirements and to trace the decision process and the rationale sustaining a software model. Also, it is not unusual that numerous technologies with similar purpose are confronted to each other during the design phase. In the present work, we propose an integrated framework combining system requirement definitions, a component-based modeling language and model transformations. Architecturally-significant requirements are explicitly linked to software architecture elements and iteratively refined or implemented by model transformations. Any transformation must be documented, even briefly, and the framework retains the transformations tree. This way, the iterative decision and design processes are completely documented for future reference or modification, i.e, designers can (i) see the mapping between a system requirement and its implementation in the architecture model, (ii) explore design alternatives or apply structural modifications without losing previous versions of the model, and finally (iii), depending on the level of documentation, at least understand partially the reasons why the model is how it is.

1 INTRODUCTION

Software systems become complex products where many people and constraints may intervene. They are intended to offer many functionalities that can evolve over time. Many possibilities are often available to fulfill specific needs which increase the amount of design choices. A requirement can be scattered over an architecture model so that it becomes difficult to recover architectural knowledge (Tyree and Akerman, 2005). Without appropriate design decisions and rationale tracing mechanisms, system maintenance, evolution and redeployment may be costly and time-consuming (Watkins and Neal, 1994).

As we present in Section 2, iterative design method in component-based systems is not a new concept. The main goal of such methods is to face requirements and constraints by integrating them step-by-step (Bosch and Molin, 1999). However, maybe the trickiest part resides in ordering these requirements since the early decisions taken at the architecture or technological levels may impact the overall design possibilities (Tang et al., 2006). Making an early decision, like choosing a particular architectural style, limits the design possibilities for later decisions. This may cause expensive rework if the decision was wrong. For example, if designers go for a *two-tier* architecture and, later on, need to replicate the database *in the cloud*, major rework at the architecture level will be necessary. Likewise, changes in the deployment infrastructure may trigger non-trivial modifications at the architecture level (Malek et al., 2012).

We propose in Section 4, an agile-based design framework intertwining structural models, requirement definitions, design rationale documentation and model transformations. These underlying languages are presented in Section 3. The aim of this work is to structure the iterative design process around step-by-step refinements and model transformations (Jansen and Bosch, 2005). On the one hand, architecturally-significant requirements are expressed regarding some guidelines. On the other hand, information systems are modeled in terms of types of constructs, concrete and interconnected instances and deployment targets. The framework traces the history of the iterative decision process with corresponding models. At any time, it is possible to go back to an earlier model, make a modification and re-apply the previously defined transformations with minor necessary rework at the architecture level. Decisions and rationale are first-class entities in the design process so that we explicitly keep the link between requirements and implementing constructs with the reasons sustaining such decisions. We do not address ordering or assessment between requirements, like the *Architecture Tradeoff Analysis Method* (ATAM) (Kazman et al., 2000) or the reasoning method proposed by Tekinerdogan *et al.* (Tekinerdogan et al., 2011), even if such methods can be integrated in our framework.

We challenged our approach on a comparative case study on a fictitious online book store system. In Section 5, we present the broad outlines of the case study and analyze some of its outcomes. We discuss how our transformation-oriented method helped designers to structure the architectural knowledge, trace design alternatives and build a documented system architecture. We will afterwards discuss the benefits and limitations of our approach in Section 6. We finally conclude this paper with our research perspectives and future work in Section 7.

2 RELATED WORK

An increasing amount of research focuses on the relations between requirements, design decisions, design rationale and architecture model. The first notable work proposed by Potts and Bruns (Potts and Bruns, 1988) records the design rationale as single entities which limits automatic extraction and reasoning between rationale types. A basis for rationale and decisions reasoning has been proposed by Kruchten *et al.* (Kruchten et al., 2006) and a formal language for decisions modeling was developed by Zimmermann *et al.* where they refined the notion of decision into *issues*, *alternatives* and *outcomes* (Zimmermann et al., 2009).

Although, there is a need for embedded facilities to maintain a concrete link between decisions and rationale, and resulting architecture models (de Boer and van Vliet, 2009). *Architecture Rationale and Element Linkage* is a more complete technique that integrates model elements and the rationale sustaining the associated design decisions (Tang et al., 2007). Jansen *et al.* introduce a documentation *enrichment* method, supported by a tool suite, to add formal knowledge even to existing documentation (Jansen et al., 2009). Zhang *et al.* propose a formal representation model for design rationale with the possibility to add a link between requirements an produced artifacts, but these products must be defined in an existing feature model (Zhang et al., 2013). However, all these approaches require modelers to maintain extra models, often with lots of mandatory details, so that the workload is significantly increased with a possible *discouraging* effect. Also, the adequacy between models is rarely ensured on the long run. In our method, decisions and rationale are kept inside requirement models, concretely linked to architecture model elements in a very simple way.

With model-driven approaches, new design methods were developed, as Rational Unified Process® (Kruchten, 2004) or the Attribute-Driven Design (Wojcik et al., 2006). Hofmeister *et al.* proposed a general model based, among others, on these two methods (Hofmeister et al., 2007). In their work, authors stated the need for an iterative design method that involves decisions and rationale as first class entities. Our method was largely inspired by these recommendations.

A couple of transformation-centric methods have emerged. In many of these approaches, models are either transformed to integrate new requirements or non functional qualities, or represent systems from a coarse-grained picture to a fine-grained one (from model to code, for example). Matinlassi proposed a technique for quality-driven model transformations where the author focuses on automation, but only on quality properties (Matinlassi, 2006). Perovich *et al* use a more complex representation for the system functionalities (Perovich et al., 2009), in terms of, among others, information flows or policies. But, as far as we know, they currently do not provide tool support and concentrate on deployment-related decisions and rationale. TransML is a family of modeling languages that provides a holistic approach for the overall design process with model transformations and verifications (Guerra et al., 2013). Its main advantage resides into the smart traceability mechanism for transformations. However, only part of the formalisms are currently implemented. In large scale projects with complex metamodels, the intrinsic complexity for writing transformation rules is not lowered and the authors give no guidelines to select the most appropriate formalism between the profusions of their languages.

3 MODELING LANGUAGES OVERVIEW

The present method focuses on software architecture design. It relies on three languages: a component-based modeling language, a requirement language with explicit design decisions and rationale traceability links, and a transformation language. In the following sections, we introduce the main concepts of both modeling languages. Afterwards, we introduce our specific transformation language in more details.

3.1 Architecturally Significant Requirement Modeling

In a previous work, we defined a simple modeling language to record and trace architecturally significant requirements (ASR) (Gilson and Englebert, 2011a), i.e, *requirements that have a significant meaning in terms of architectural artifacts* (Clements and Bass, 2010). We provide in Listing 1 a sample requirement model of a Client-Server system. Two requirements are listed: a functional requirement identified by the name *SayHello*, and a non functional one named *FastAnswer*. They are both assigned to the same model object `Server`. We use fully qualified names with single dots as delimiters to unambiguously identify models and constructs.

```
// model header (package name)
package example;
// asr model name linked to an architectural model
asrmodel clientserver with example.clientserver{
   //assignation of a requirement to a component
   func SayHello assigned Server{
   long description "The Server shall print 'Hello World!' to the console.";
   //design decision type: a transformation set
    realisation example.myfirsttransformation;
    // rationale for this design decision
    rationale{
      assessment"Functionality is trivial, a unique service should make the trick.";
      strength"Very simple implementation with unique service without parameters.";
      weakness"The printed message is fixed.";
    }
  }
  // non functional requirement
  nonfunc FastAnswer assigned Server {
   long description "When receiving a client request, Server shall answer in less than 1 second.";
   // fulfill ASR by implem. "Hello" interface
    implements Hello;
    rationale{
      assessment"With parameter-less service, server response time should be fast.";
      assumption"Because service is trivial, a unique server should be sufficient.";
      constraint"Should be less than 100 simultaneous requests per second";
    }
  }
}
```

Listing 1: Sample ASR model for a Client-Server.

The present ASR model `clientserver` is part of a package and refers to an architecture model named `example.clientserver`. Both requirements are described following the writing guidelines from Alexander and Stevens (Alexander and Stevens, 2002) and conforms to the EARS templates (Mavin and Wilkinson, 2010). In short, these templates define a structured way of writing system requirements in natural language. Specific pieces of information, like events, or options, are highlighted by specific terms, respectively *when* and *where*, and are present in the description at specific places. Amongst the advantages of this approach, we particularly note its ease of learning since no new (modeling) language or concept is necessary to learn, as well as its ability to induce more completeness and conciseness in requirement descriptions.

Regarding a requirement, a number of decisions can be taken. We group them in the following categories:

Assignation: the requirement is assigned to a modeling construct.

Refinement: a lower-level requirement is a refinement of a higher-level one, i.e., concerns part of the scope of the higher-level requirement, but describes it more precisely.

Alternative: a lower-level requirement is a possible refinement alternative for a higher-level requirement.

Selection: an alternative is actually selected by the designers as the *implementation* solution.

Interface Usage or Implementation: in order to fulfill a requirement, an existing interface is used or implemented, or in case of non functional requirements, the given interface conforms to the needed properties to achieve this requirement.

Re-assignment: the requirement is reassigned to another modeling construct, i.e, the responsibility to accomplish the requirement is transferred to another model element (mainly software components).

Realisation: a structural modification must be made into the component model and this will be expressed as a model transformation (cfr. Section 4).

When a modeler takes a decision, like writing a model transformation to implement it in the architecture model, the decision type is recorded in the model. For any type of decision, a set of rationale can be added of which the *assessment* is mandatory. We briefly present here the type of rationale that can be filled in an ASR model.

Assessment: the actual reason sustaining the design decision; this is the only mandatory piece of information.

Assumption: any assumption made on the environment or on other model elements.

Strength: any advantage of this decision.

Weakness: any disadvantage or limitation of the decision.

Constraint: any constraint under which the decision is taken, or a further consequence related to this decision.

The proposed syntax for ASR models enforces designers to document their choices in terms of assignments of requirement to modeling constructs, and in terms of refinements of requirements. First, we explicitly trace the link between a requirement and the model element in charge of its implementation. This enhances the architectural knowledge regarding *who* is implementing *what*. Second, we keep the history of the decisions regarding a requirement (reassignments, refinements and alternatives) for documentation purposes. Third, design decisions must be documented by at least one reason sustaining such a choice. A minimum amount of information is mandatory in order to avoid putting to much unnecessary or unwanted effort in documentation tasks. Further details, like strengths and weaknesses, can be added by the modeler at his own discretion to justify his choices more explicitly.

3.2 Architecture Modeling

Attached to a requirement model, a structural definition of the system must be provided. For this purpose, we defined a *3-level* component-based modeling language (Gilson and Englebert, 2011b). We present in the following how information systems are modeled in three inter-related stages: *definition*, *assemblage* and *deployment* (DAD). Wider definitions and justifications of the language constructs are presented in the aforementioned paper.

3.2.1 Stage One: Definition

Roughly, in the first stage, abstract component types are connected by link types through interfaces. The model can be at any level of details and component types can contain other types connected by inner interfaces. For instance, a first architectural representation could be composed by only one component named `System` with all requirements assigned to it. Listing 2 illustrates part of the *definition* stage for our Client-Server.

```
package example;
dadmodel clientserver {
  definition {
  interface Hello {
    sync void hello ();
  }
    componenttype Client {
    uses Hello as hello;
    }
    componenttype Server {
    implements Hello as hello;
    }
    connectortype One2One {
    mode one2one;
    }
  linkagetype from Client.hello to Server.hello
        with One2One;
  }
}
```

Listing 2: Definition stage for a Client-Server.

As for an ASR model, a DAD model must belong to a package. In this case, we decided to use the same names for both ASR and DAD models for convenience. We defined a simple interface `Hello` with a unique synchronous service without parameters named `hello()`. This interface is used by a type of component `Client` and implemented by a type of component `Server`. When an interface is exposed in any way by a type of component, it becomes a facet of this component with a given polarity (usage or implementation). We also define a type of connector which is point-to-point, i.e. connecting one type of component to only one other type of component at a time. We finally link the `Client` to the `Server` with the `One2One` type of connector in a *provide-require* contract through the facets.

A set of primitive types has been defined, as integer, boolean or string. Architects may obviously define new primitive types or custom data structures themselves. In a DAD model, a primitive type, a structure or an interface are all considered as generic types and can be used to type a parameter.

At this point, we defined the building blocks we can *instantiate* and concretely connect during the *assemblage* stage. The type of linkage defined from now only constrains how the type of components can be linked to each other (i) through which interface and, (ii) how many instances of a type of a component will be involved in the connection.

3.2.2 Stage Two: Assemblage

Now we specified these types of blocks, we can define a concrete architecture by instantiating and connect them with concrete links. In Listing 3, we introduce a communication protocol and complete the definition of the type of connector, then we present the *assemblage* stage of the DAD model.

```
definition {
  //omitting previously defined constructs
  //communication protocol for concrete binding
  protocol TCP {
    layer: transport;
    reliable: true;
    ordered: true;
    secured: false;
    mode: one2one;
  }
  // link support protocol (same layer)
  connectortype One2One {
    mode one2one;
    layer: transport;
    accepts TCP;
  }
}
assemblage {
  // a set of instances of type Client
  soi client[0 100] : Client ports {
   //exposing the required facet hello over TCP
    Client.hello as hello on TCP;
  };
  // a set of instances of type Server
  soi server : Server ports {
   //exposing the provided facet hello over TCP
    Server.hello[20] as hello on TCP;
  };
  // a link of type One2One
  connector con : One2One;
  //concrete binding from clients to the server
  linkage from client.hello to server.hello
       with con;
}
```

Listing 3: Assemblage stage for a Client-Server.

The protocol TCP specifies with basic, but extensible properties, a communication protocol that will be used to support the connection between instances of component types. A protocol is defined, among others, by the communication layer enabling to specify a wide range of connection protocols from low-level protocols like Bluetooth to high-level ones like Telnet or Java method call. In our case, we use the TCP protocol and add it to the list of accepted protocols for our connector type.

We create a set of instances (SoI) for each component type previously defined. SoI are declared with a minimum and a maximum cardinality that express the amount of instances of the same type that can be present in a concrete architecture. In our example, a maximum of 100 instances of the `Client` component type can be created. This SoI has one port typed by the facet `hello` on the `TCP` protocol. The `Server` is unique and has 20 ports of the type `hello` available, also over `TCP`. `Clients` SoIs are linked to the `server` according to the linkage type defined at the previous stage. At this point, we specify a concrete architecture instance with a certain amount of each component types, available ports (i.e., interface instances) and connections on specific protocols.

3.2.3 Stage Three: Deployment

As presented in Section 1, a notable deficiency in current design methods is the omission of infrastructure constraints. We believe that this problem can be partially tackled by integrating the constraints as soon as they appear in the design phase. We provide basic and extensible building blocks to define the target infrastructure. In Listing 4, we specify these types of blocks and illustrate the abstract *deployment* phase.

```
definition {
  // omitting previously defined constructs
  // type of physical port
  gatetype Ethernet {
    supports TCP;
  }
  // type of computation node (machine)
  nodetype Computer gates {
    Ethernet eth;
  }
  // type of concrete (physical) link
  mediumtype E100BaseT {
    supports TCP;
  }
}
assemblage { /* hidden for conciseness */ }
deployment {
  // 101 nodes of type Computer
  node computer[101] : Computer;
  // cable plugged into computers ethernet
       gates
  plug E100BaseT into computer[0 100]::eth;
  // clients deployed on computers from 0 to 99
  deploy client on computer[0 99];
  // the server is deployed on the 100th
       computer
  deploy server on computer[100];
  // the ports typed by the Hello interfaces
       accessible from the ethernet gates
  open client.hello on computer[0 99]::eth;
  open server.hello on computer[100]::eth;
}
```

Listing 4: Deployment stage for a Client-Server.

Three new types of model elements are created at the *definition* stage. First, a type of gate specifies a network interface or a physical port on a computation device. Gate types support a possibly non exhaustive list of protocols. In our example, we define an `Ethernet` gate type supporting our `TCP` protocol. Second, we define a type of node that can represent any type of computation machine. A node type can be equipped by a number of gates of certain types. Third, type of communication media, such as network cables, are defined. A type of medium also support a list of protocols. These physical infrastructure-related constructs can be more precisely defined by an extensible property mechanism currently under development and out of the scope of this paper.

When we have specified types of nodes, gates and media, we can define an abstract deployment by mapping the set of instances onto these physical constructs. For this purpose, we have to create 101 nodes of type `Computer` (remember we can have up to 100 `Client` instances and only one `Server` instance). We do not need to specify medium instances. Since we are only concerned by the properties attached to a type of medium used in the target infrastructure and how nodes are accessible from *outside*, we abstractly *plug* communication medium types into the gate from a node.

We now *deploy* our set of instances on nodes and *open* their ports (typed by interfaces) on gates. The overall communication binding, from abstract templates defined at the *definition* stage, to the physical connections specified here at the *deployment* through concrete links over ports at the *assemblage*, is *reified* using the communication protocol. In this example, we use a unique protocol (`TCP`), but more complex verifications with *compatible* protocols can be done, depending on user-defined properties.

Note that, in a DAD model, all stages are optional. One can isolate whatever he wants in a specific DAD model and import (using the `import` keyword in the model header) any other model element defined elsewhere. This mechanism will be further developed in Section 4.4 when we will talk about pattern injections.

4 STEP-BY-STEP REFINEMENT WITH MODEL TRANSFORMATIONS

Much research has emerged to structure the architecture design method in an iterative way. We describe here how the aforementioned languages are used in our design framework.

During the creation of the architecture of an information system, architects usually start from a coarse-grained architectural style, choose one architecturally significant requirement (regardless how they prioritize them), refine it to more precise ones if necessary and *implement* it in the architecture. At some points in the design process, architects performs some validation of the produced model(s). During this iterative process, it often happens that multiple alternatives are explored or that wrong decisions have been taken so that architects have to backtrack to a previous version of the model. As pointed out in Section 1, this decision process and the rationale sustaining a *final* architecture are frequently lost after a while and further evolutions and bug fixes become time consuming or error-prone.

The approach used in our framework is transformation-centric: any change in an architecture model must be expressed as a model transformation. In order to test our proposal, we implemented both languages presented in Sections 3.1 and 3.2 and the transformation language described in the following as Eclipse plugins with the Xtext[1] framework. Xtext is a toolset to build configurable textual editors for a DSL as Eclipse plugins on top of the Eclipse Modeling Framework[2]. We also implemented a transformation engine working on the abstract syntax tree of our model in the Xtend[3] language. In short, Xtend is an extension of Java, fully compatible with Java, introducing lambda expressions among other facilities, and compiled into Java code.

From the definition of all known requirements in an ASR model and a first, even empty, DAD model, any modification to the architecture must be expressed using one of the following transformation rules. A single file can group many transformations, related to one requirement, that will be applied all in once. A new model is then created by a transformation engine and this model can be further refined, i.e. transformed, to implement other requirements.

We group rules related to a specific requirement in a DAD-Transformations set (DAD-T). Before going into the description of the existing rules, we present in Listing 5 the general template of a DAD-T set.

Similarly to DAD and ASR models, a transformations set belongs to a package. A transformation is always related to a requirement, identified in the model by the keyword `concerns` defined in the linked ASR model declared on top of the DAD-T file. A DAD model can also be referenced when the transformation is linked to a specific architecture model, i.e.,

[1] www.eclipse.org/Xtext

[2] http://www.eclipse.org/modeling/emf/

[3] http://www.eclipse.org/xtend/

```
package example;
// involved asr model (mandatory)
asrmodel example.clientserver;
// involved dad model (optional)
dadmodel example.clientserver;
// transformations set name refering to an ASR
transformationset template concerns SayHello {
  /* there should be some rules in here */
  assemblage {/* some new assemblage */}
  deployment {/* some new deployment rules */}
}
```

Listing 5: Template DAD-T set.

the transformations set is not a definition of a pattern (more details about patterns in Section 4.4). In this case, the file contains a set of rules to alter the referenced architecture model and, eventually, define new *assemblage* or *deployment* statements. Hereafter, we present the available transformation rules.

4.1 Creation of Constructs

At some point in the design process, new components, interfaces or communication links should be specified. A creation rule is defined by the keyword `create` followed by the definition of the new construct. In Listing 6, we show how the `Hello` interface could have been created with related `Client` facet and port in a transformation model.

```
// create interface
create interface Hello {
  sync void hello();
}
// create facet in Client component type
create facet {
  uses Hello as hello;
} in Client;
// create port in client set of instance
create port {
  Client.hello as hello on TCP;
} in client;
```

Listing 6: Creation of the `Hello` interface.

Any other construct from the *definition* stage can be created in a similar fashion, same as for ports (from the *assemblage* stage). The transformation engine will inject the newly created constructs into the bound DAD model. Prior to a creation, the engine verifies that no name conflict occurs: names must be unique by construct type. A check is performed using fully qualified names to ensure name uniqueness for a specific type of construct.

4.2 Deletion of Constructs

It is indeed possible to delete constructs from an architecture model. Not only any element defined at the *definition* stage can be deleted, but also any other construction at the other levels, such as sets of instances, gates, linkages or deployment statements.

The transformation engine ensures that deletions are always done *in cascade*, i.e., all related constructs or instances are deleted when a particular element is deleted[4]. For example, Listing 7 is the resulting model after the deletion of the `Client` component type from the model illustrated in Listing 4.

```
definition {
  interface Hello {
    sync void hello ();
  }
  /* component type Client deleted */
  componenttype Server {
    implements Hello as hello;
  }
  /* omitting remaining of def. stage */
}
assemblage {
  // set of instance typed by Client deleted
  soi server : Server ports {
    Server.hello as hello on TCP;
  };
  connector con : One2One;
  /*linkage with the soi "client" is deleted*/
}
deployment {
  node computer[101] : Computer;
  plug RJ45 into computer[0 100]::eth;
  /*deployment of the soi "client" is deleted*/
  deploy server on computer[100];
  /* gate opening deleted too */
  open server.hello on computer[100]::eth;
}
```

Listing 7: Deletion of `Client` component type.

In this example, the related linkage type involving the `Client` component type has been removed. The transformation engine deleted the set of instances typed by the `Client`, the linkage where this set of instances appeared, as well as the related deployment statements. A similar behavior is always applied for all other model elements where the engine removes all constructs with a reference to the suppressed element.

[4]For a complete view of the relations between model constructs, please refer to (Gilson and Englebert, 2011b).

4.3 Fine-grained Alteration of Constructs

Frequently, architects need more fine-grained transformations where they can, for example, add a service to an interface, add a gate to a node or alter a data structure. For example, in Listing 8, we modify the `hello` interface.

```
alter interface Hello {
  // add new asynchronous service with a delay
  // before saying HelloWorld
  add async invokeHello(int delay);
  // add an input parameter to specify the message
  rewrite hello {
    add in string message;
  }
}
```

Listing 8: Fine grained alteration of an interface.

A new asynchronous service is added into the interface with a parameter that specifies the delay before invoking the `hello` service. The second alteration adds a new parameter to the service to define the content of the message. Note that for synchronous services, we have to specify the access type of a parameter (*input*, *output* or *both*).

For every construct with *internal* definitions, like types of nodes or data structures, similar transformations can be defined. Here again, name validity checks are performed to avoid conflicts.

4.4 Pattern Definition, Injection and Replacement

In order to enhance re-usability, design patterns can be created in our approach. They are expressed as transformations sets, as illustrated in Listing 9, and linked to an ASR model specifying their assets (not presented here, because of lack of space).

```
package example;
asrmodel example.observer;
transformationset observer concerns Observer {
  create interface IObserver {
    async notify();
  }
  create interface ISubject {
    async register(IObserver o);
    async unregister(IObserver o);
  }
  create componenttype Observer {
    implements IObserver as iobserver;
    uses ISubject as isubject;
  }
  create componenttype Subject {
    implements ISubject as isubject;
    uses IObserver as iobserver;
  }
  create connectortype Simple {mode one2one;}
  create connectortype Multi {mode one2many;}
  create linkagetype from Subject.iobserver to Observer.iobserver with Multi;
  create linkagetype from Observer.isubject to Subject.isubject with Simple;
}
```

Listing 9: Definition of the `Observer` pattern.

The transformations set creates the two necessary type of components (`Observer` and `Subject`) as well as the interfaces, facets and types of linkage. A pattern must be self-contained, i.e. all needed constructs are defined in the model, or it can import some external resources with the `import` keyword, similarly to DAD models.

```
package example;
asrmodel example.clientserver;
dadmodel example.clientserver;
transformationset inj_observer concerns Observer {
  // execute pattern transformation
  include example.observer;
  // merge definition of Subject into Server
  replace Subject by Server merge;
}
```

Listing 10: Pattern injection with merge.

To inject this pattern into an architecture model, assume we had a requirement asking for such a pattern, we simply need to *include* the pattern-related transformations set, then to *replace* and *merge* the target constructs, as shown in Listing 10. The `include` mechanism can be used for any type of reusable transformation. Note that if the `merge` option is not passed, a construct is totally *replaced* by another one without *merging* their definitions.

The result is presented in Listing 11. An `Observer` construct has been created and the `Server` now implements and uses the `Subject`-related interfaces. More complex replacements with specific *overrides* can also be defined. For example, one can override a given facet by another with a compatible definition (i.e., services signatures and properties).

Other types of transformations exist, but are not shown here because of lack of space such as renaming, moving elements, etc. For example, an inner component type can be moved to another one or raised up to the root of the model.

```
definition {
  interface IObserver {/* omitting definition */}
  interface ISubject {/* omitting definition */ }
  componenttype Server {
    implements Hello as hello;
    implements ISubject as isubject;
    uses IObserver as iobserver;
  }
  componenttype Observer {
    implements IObserver as iobserver;
    uses ISubject as isubject;
  }
  connectortype Multi {/* omitting definition */}
  connectortype Simple {/* omitting definition */}
  linkagetype from Server.iobserver to Observer.iobserver with Multi;
  linkagetype from Observer.isubject to Server.isubject with Simple;
  // omitting remaining of file
}
```

Listing 11: After pattern injection.

4.5 Decision and History Tracking

Coupled with the rationale and design decision tracing in ASR files, as presented in Section 3.1, we implemented a simple tree-based history tracing mechanism. Figure 1 shows a screen capture of our Eclipse plugin and illustrates a history tree in the package explorer (left-hand side of the picture).

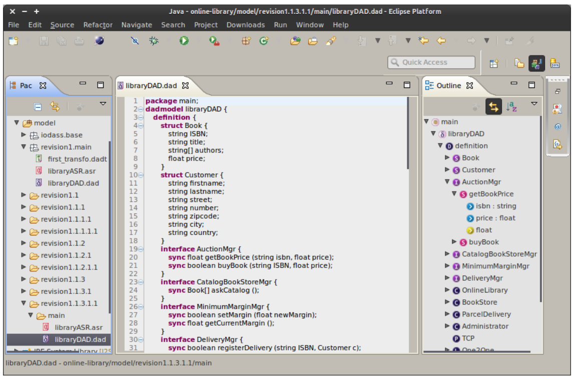

Figure 1: Capture of the Eclipse plugin with history tracking in package explorer.

Every time a model is transformed, a new model is created in a separate folder. By convention, we start from a folder named `revision1` and append `.1` after a transformation is carried on a DAD model. When backtracking to a previous model, a new branch will be created by increasing the higher sub-branch number. For example, after the model `revision 1.1.1.1`, we started back from the model at revision `1.1` so a new branch numbered `1.1.2` was created.

This simple mechanism keeps the history of all created models and can be used to draw a graphical tree with models as nodes and transformations as edges for more convenient history browsing.

We intend to review the iteration process in order to define *release points* where a set of transformations are grouped to define coarse-grained evolutions or patches, which make more readable models by humans and offers possibilities for software configuration management.

Other avenues worth exploring would be to see how task-oriented methods, like the *MyLyn*[5] project, can be integrated into our framework, as well as collaborative model editing and versionning (Rutle et al., 2009).

5 A COMPARATIVE CASE STUDY

We experimented our approach with a comparative case study on a fictitious online book store system. We confronted our framework against SysML (Object Management Group, 2010). In brief, a web-based online library presents a catalog of books which is the aggregation of the catalogs of a set of book stores. When a customer purchases a book, the library starts an auction between the book stores to buy the book at the cheapest price. After, the library contacts a delivery system to pick up the book at the winning book store and to deliver it to the customer.

The case study was conducted on a group of 24 master students at the University of Namur, all familiar with UML diagrams, but not with SysML, neither with our framework. We organized a preliminary round to evaluate their system modeling competences. During a lecture, the students were asked to draw a class diagram based on a 12-pages requirement document. The document contained a precise description of use cases for a simplified vehicle inspection system. Three researchers, also familiar with software modeling, classified the diagrams in four categories based on their syntactic and semantic correctness. This way, we divided the students in two groups of comparable competences and made teams of two students for the remaining of the study, the first group had to design the online library in SysML, the second with our framework.

[5]www.eclipse.org/mylyn/

We presented separately the language artifacts and the tool support to each group, so that the first group knew only about SysML and the second only about our languages. Both groups also received the same description of the *system-to-be* and were asked to build it (design and code) in two phases. For the first phase, the requirements were clearly stated in the documents to let them getting familiar with the new languages. For the second phase, the descriptions were more fuzzy and were related to the evolution of the system.

After each phase, we evaluated the quality of the models and the documentation created by the students, as well as the functional correctness of the produced code. We verified to what extend they documented the decisions sustaining the produced architectures. Basically, we checked if all model elements had at least one justification explaining why they were created into the model.

After the second phase, students were asked to answer to a questionnaire in classroom to evaluate the expressiveness, the documentation and evolution facilities of the languages they used. The students also formulated reviews and advices in a document where they could express their feelings regarding the aforesaid criteria. In the questionnaire, we used a non-graduated ruler going from “fully disagree” (**0** value) to “fully agree” (**5** value) and measured the students answers. We also dissimulated redundant questions in order to double-check the given answers. On the 24 questionnaires, we discarded four of them for each group because the gaps between the answers to these control questions were too large.

We present in Table 1 part of the results regarding expressiveness, evolution and documentation facilities. The second column (**S**) shows the aggregated rating for SysML and the last column for our framework (**D**).

As shown in the Table 1, our framework offers a significant improvement regarding constructs expressiveness (question 1). Regarding documentation (question 2), the difference is not significant enough to state that our framework gives better results. Concerning model evolution (question 3), the results of our framework are slightly inferiors. Two aspects can explain these last values. First, our framework relies on textual models which are, by nature, less visual than graphical models. Second, during the experiment, due to a bug, the comments present in the DAD models were lost after a transformation, which revealed to be a significant lack at participants eyes when we analyzed their reviewing document.

In their reviews, most of the students noted the requirement traceability improvements offered by our framework as well as the ease of justifying the design rationale and decisions compared to their experience (mainly with UML diagrams and text-based documentation).

Table 1: Sample questions and aggregated ratings.

Questions	**S**	**D**
1. The languages constructs allow to represent:		
a. the functionalities of the system.	3.68	4.33
b. the technological and communication constraints.	1.95	4.48
c. the non functional requirements	3.10	4.12
2. The written documentation allows to efficiently comprehend the system within the framework of a modification of the system.	2.63	2.85
3. During the second phase:		
a. a major work was necessary to comprehend again the architectural concepts of the system.	0.97	1.21
b. the modeling languages eased the structural changes linked to the new functionalities to implement.	3.83	3.25

6 DISCUSSION AND LIMITATIONS

The ASR model covers a notable part of the project backlog, as defined by Hofmeister *et al.* (Hofmeister et al., 2007), which is a key document for system engineering. Design rationale, decisions and structured requirements are present in the model and related to the modeling constructs that implement them. This simple mechanism enhances the architectural knowledge without asking much documentation effort. It can be easily extended to add *meta-information* regarding the project itself, like requirement ordering methods, standards and so forth.

In terms of the analyzing template for architecture design methods presented by Hofmeister *et al*, the proposed framework addresses the questions related to the produced artifacts and the tools support. Regarding producing activities, the framework enforces documentation of design iterations and records changes in the architecture model as explicit model transformations.

The transformation inclusion mechanism offers a lightweight way to define and re-use patterns in a transformations set. Working on the concrete syntax results in more concise rules and improves the readability of such transformations comparing to general-purpose transformation languages. However, reusability of transformations is limited to concrete syntax elements and patterns injections require to write mapping rules between pattern constructs and the current model elements.

Coupled with the history mechanism, design or technological alternatives can be explored and documented. Though, a proper graphical visualization facility should be provided to efficiently identify *deltas* between models and to navigate easily between revisions. This way, our framework can be used for software configuration management where patches and redeployment rules are expressed by model transformations.

Also, a couple of relations between design decisions should be taken into account, like conflicts between requirements or other natures of impacts, in order to enhance the decision-making process for designers. Furthermore, we should evaluate how a structured formalism can be integrated into the requirement models to define constraints for validation and simulation purposes.

At current point of development, models are expressed in textual syntax. Even if such representation is very expressive, the analysis and communication of textual models is often less natural for humans. The underlying Xtext framework is compatible with EMF-based graphical tools and allows to synchronize both representations on the same model.

Last, a larger case study should be conducted in order to evaluate if the transformation-centric method coupled to a task-oriented approach scales to industrial cases.

7 CONCLUSIONS AND FUTURE WORK

We introduced a transformation-centric design framework based on Domain Specific Languages. Architectural constructs are explicitly related to requirement specifications and implemented iteratively in the architecture with model transformations. Every decision is recorded in the requirement model with its design rationale. A tool is provided for textual models, as well as a transformation engine. We conducted a comparative case study to partially validate our proposal and evaluate its benefits.

In the future, we intend to add behavioral specifications to component types, interfaces and communication protocols. As a first step, tag-based properties and logical constraints facilities will be added to further specify modeling elements. With behavioral properties, designers will be able to ensure a transformation does not break behavioral aspects of an existing architecture.

A Java code generator is under development. At present time, Java classes and interfaces are generated from DAD models with method signatures. Primitive types can be mapped to existing Java types or generated as separate classes. We intend to maintain a bi-directional link between architectural models and Java code for co-evolution purpose.

A visual representation of the history tree with model *deltas* should be integrated in the tool to facilitate further references to transformations outcomes, design alternatives and system evolutions. Ideally, a final goal would be to synchronize a graphical representation on textual architecture models to benefit from the advantages of both textual and graphical visualizations.

REFERENCES

Alexander, I. F. and Stevens, R. (2002). *Writing Better Requirements*. Addison-Wesley.

Bosch, J. and Molin, P. (1999). Software architecture design: Evaluation and transformation. *IEEE Int. Conf. on the Engineering of Computer-based Systems*, pages 4–10.

Clements, P. C. and Bass, L. (2010). Relating business goals to architecturally significant requirements for software systems. Technical Report CMU/SEI-2010-TN-018, Soft. Eng. Institute, Carnegie Mellon University.

de Boer, R. C. and van Vliet, H. (2009). On the similarity between requirements and architecture. *J. Syst. Softw.*, 82(3):544 – 550.

Gilson, F. and Englebert, V. (2011a). Rationale, decisions and alternatives traceability for architecture design. In *Proc. of the 5th European Conf. on Software Architecture, Companion Volume*, page 4. ACM.

Gilson, F. and Englebert, V. (2011b). Towards handling architecture design, variability and evolution with model transformations. In *Proc. of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 39–48. ACM.

Guerra, E., Lara, J., Kolovos, D. S., Paige, R. F., and Santos, O. (2013). Engineering model transformations with transml. *Software & Systems Modeling*, 12(3):555–577.

Hofmeister, C., Kruchten, P., Nord, R. L., Obbink, H., Ran, A., and America, P. (2007). A general model of software architecture design derived from five industrial approaches. *Journal of Systems and Software*, 80(1):106 – 126.

Jansen, A., Avgeriou, P., and van der Ven, J. S. (2009). Enriching software architecture documentation. *J. Syst. Softw.*, 82:1232–1248.

Jansen, A. and Bosch, J. (2005). Software architecture as a set of architectural design decisions. In *Proc. of the 5th Working IEEE/IFIP Conf. on Software Archi-*

tecture, pages 109–120, Washington, DC, USA. IEEE Computer Society.

Kazman, R., Klein, M., and Clements, P. (2000). ATAM: method for architecture evaluation. Technical Report CMU/SEI-2000-TR-004, Soft. Eng. Institute, Carnegie Mellon University.

Kruchten, P. (2004). *Rational Unified Process: An Introduction*. Addison Wesley Professional, 3rd edition edition.

Kruchten, P., Lago, P., and Vliet, H. V. (2006). Building up and reasoning about architectural knowledge. In *Proc. of the 2nd Int. Conf. on the Quality if Software Architectures*, pages 43–58.

Malek, S., Medvidovic, N., and Mikic-Rakic, M. (2012). An extensible framework for improving a distributed software system's deployment architecture. *Software Engineering, IEEE Transactions on*, 38(1):73–100.

Matinlassi, M. (2006). *Quality-driven software architecture model transformation: Towards automation*. PhD thesis, ESPOO: VTT technical Research Centre of Finland. VTT Publications 608.

Mavin, A. and Wilkinson, P. (2010). Big ears (the return of "easy approach to requirements engineering"). In *Proc. of the 18th IEEE Int. Requirements Engineering Conf.*, pages 277–282.

Object Management Group (2010). *OMG Systems Modeling Language, version 1.2*.

Perovich, D., Bastarríca, M. C., and Rojas, C. (2009). Model-driven approach to software architecture design. In *ICSE Workshop on Sharing and Reusing Architectural Knowledge*, pages 1–8.

Potts, C. and Bruns, G. (1988). Recording the reasons for design decisions. In *Proc. of the 10th Int. Conf. on Software Engineering*, pages 418 –427.

Rutle, A., Rossini, A., Lamo, Y., and Wolter, U. (2009). A category-theoretical approach to the formalisation of version control in mde. In *Fundamental Approaches to Software Engineering*, volume 5503 of *Lecture Notes in Computer Science*, pages 64–78. Springer Berlin Heidelberg.

Tang, A., Babar, M. A., Gorton, I., and Han, J. (2006). A survey of architecture design rationale. *J. Syst. Softw.*, 79(12):1792 – 1804.

Tang, A., Jin, Y., and Han, J. (2007). A rationale-based architecture model for design traceability and reasoning. *J. Syst. Softw.*, 80(6):918–934.

Tekinerdogan, B., Özturk, K., and Dogru, A. (2011). Modeling and reasoning about design alternatives of software as a service architectures. In *Proc. of the 9th Working IEEE/IFIP Conf. on Software Architecture*, pages 312–319.

Tyree, J. and Akerman, A. (2005). Architecture decisions: Demystifying architecture. *IEEE Software*, 22:19–27.

Watkins, R. and Neal, M. (1994). Why and How of Requirements Tracing. *IEEE Software*, 11:104–106.

Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R., and Wood, B. (2006). Attribute-driven design (ADD), version 2.0. Technical Report CMU/SEI-2006-TR-023, Soft. Eng. Institute, Carnegie Mellon University.

Zhang, Y., Luo, X., Li, J., and Buis, J. J. (2013). A semantic representation model for design rationale of products. *Advanced Engineering Informatics*, 27(1):13 – 26.

Zimmermann, O., Koehler, J., Leymann, F., Polley, R., and Schuster, N. (2009). Managing architectural decision models with dependency relations, integrity constraints, and production rules. *J. Syst. Softw.*, 82(8):1249–1267.