Towards Handling Architecture Design, Variability and Evolution with Model Transformations

Fabian Gilson and Vincent Englebert
PReCISE Research Center
Faculty of Computer Science
University of Namur, Belgium
{fgi,ven}@info.fundp.ac.be

ABSTRACT

Software systems have to face evolving requirements from information system stakeholders, infrastructure modifications, and evolving rationales about the implementation. This increases the rate of migration and redeployment of systems. Recent approaches intend to abstract architectural element specifications from the implementing technology and manage software design through model transformations. Based on an Architecture Description Language integrating infrastructure modelling facilities and a requirement modelling language, the present work manages architecturally significant requirements and infrastructure evolutions by model transformations. Our approach offers support for evolution and variability management tasks as it makes explicit the rationales concerning requirements, infrastructure and implementation alternatives that guide both the software architecture and the infrastructure definition.

Categories and Subject Descriptors
D.2.10 [Design]: Methodologies; D.2.11 [Software Architecture]: Language; D.2.2 [Design Tools and Techniques]: Computer-aided software engineering (CASE)

General Terms
Design, Languages

Keywords
architecture description language - architecturally significant requirement - infrastructure constraint - architecture variability - model transformation

1. INTRODUCTION

Handling the software’s variability and evolution are major challenges in computer science. On one hand, component-based information systems (IS) are built upon a set of required and optional features. On the other hand, they have to integrate new requirements and migrate rapidly from one technology to another. IS are redeployed frequently to changing physical infrastructure since new technologies appear frequently. Therefore, engineers can use models to build abstract views of systems and, this way, ease the understanding and abstract manipulation of these systems.

Since IS are growing in size and complexity, as well as they are interacting with each other, architecture description languages (ADL) are focusing on a high-level representation of systems. These languages often offer encapsulation facilities in order to represent the same system from a coarse-grained abstraction level to a fine-grained one. This eases the comprehension and complexity handling of large systems. However, few ADLs integrate transformation languages and facilitate the validation of architecture models against the constraints of the target infrastructure. Moreover, architecture variability is anecdotally considered by such approaches.

The present work defines a transformation-oriented approach for distributed information systems design. We define an architecture description language coupled with an architecturally-significant requirement formalism. By coupling the architecture and requirement views, we first intend to give support to systems evolution tasks as well as offering design rationale traceability. Second, we want to link implementation alternatives and corresponding architecture models. Third, we want to provide infrastructure-related constructs that will be used to validate the deployment with an abstract representation of the targeted infrastructure.

The Section 2 lists the related work. Afterwards, the main concepts of our proposed ADL is presented in Section 3. Then, we talk about our requirement modelling language in Section 4. Next, in Section 5, we propose a design method for component-based information systems, based on the aforementioned languages. In Section 6, we illustrate our proposal with the example of a distributed image renderer. We discuss the pros and cons of our approach in Section 7. In Section 8, we expose the limitations of the approach. We present the future work in Section 9. Last, we conclude in Section 10.

2. RELATED WORK

Many architecture description languages (ADL) have been proposed to aim to represent complex information systems. A set of these languages has been compared regarding criteria like extensibility, non-functional requirements definition and evolution possibilities [21]. None of the languages compared in this report offered all of these assets. More recently,
the Architecture Analysis and Design Language (AADL) [9] answered to part of these lacks, but focuses mainly on embedded systems architecture models. AADL models are really close to the implementation artefacts, missing abstraction possibilities. All these ADLs do not integrate model transformation facilities, so practitioners have to use dedicated tools to manipulate their models. As in our proposal, SafArchie [2] includes a transformation language. However, connections are only synchronous procedure calls and model constructs semantics cannot be refined by non-functional properties. Our proposal offers richer connection possibilities as well as extensible constructs.

Many parties are potentially involved in component-based architecture and such systems are often deployed on heterogeneous existing hardware. It is then crucial to build an abstract representation of the infrastructure and to model the deployed artefacts. Few methods offer facilities to abstractly deploy architecture models and they provide only limited semantics. Object Management Group (OMG) UML deployment diagrams [26] is one of those. Its construct semantics is very limited, especially for communication links, because it has been thought generic. With the Deployment & Configuration standard [24], the OMG created a richer language, but is too closely coupled to the CORBA middleware [25] and then makes hard the mapping to other component-based languages. SafArchie also integrates abstract deployment facilities, but the construct semantics is limited and cannot be extended by non-functional requirements, unlike our proposal.

Software systems have to cope with changes in requirements as nowadays environment is evolving rapidly [29]. Recent approaches have been specified to formalise the design rationale and design decisions knowledge. One of the most crucial requirements of such approaches is the binding between the software architecture and its design choices [4, 32]. An ontology of design decisions for software intensive systems is presented in [18]. A decision tree modelling language built on top of this ontology is proposed in [17]. In [33], the authors introduce a formal language for decision modelling. They refine the notion of decision as presented in [17] into alternative, outcome, issue, etc. Also, they can define a more complete set of relations between decisions. At present time, we reuse a limited part of the latter’s work and concretely bind architecture and requirement models.

A systematic review of variability management in software product line has been proposed in [8]. A couple of work addressed by the authors focus on system architecture. Most of these, as well as the work proposed in [10] for service-oriented distributed systems, define variability on systems with variation points on the architecture models themselves. Our approach creates separate models for each alternative. Variation points offer more flexibility and enable runtime variability support, but decrease the readability. Also, by separating alternative requirements or design decisions and their implementing artefacts from the architecture model, recurrent solutions can be more easily extracted and reused.

3. ARCHITECTURE DESCRIPTION LANGUAGE

3.1 Overview

As showed in Section 2, existing architecture modelling approaches have significant lacks regarding our needs of evolution, variability, abstract deployment and traceability capabilities. Therefore we introduce DAD, an ADL covering these features. We represent architecture models in three steps called Definition, Assemblage and Deployment. We provide a simple Client-Server sample in Figure 1 as an illustration.

![Figure 1: Definition-Assemblage-Deployment sample model](image)

The Definition stage specifies abstract component types, namely the Client and the Server. Functionalities of the component types are expressed as method signatures that are grouped in interfaces. The component types are linked by a provide-require contract, i.e., the Facet interface. The Client requires the Facet interface provided by the Server. The connection between the Client and the Server is supported by a connector type constrained by a set of accepted communication protocols.

Component types are incarnated as sets of instances that express the minimum and maximum number of instances of a component type that can be present at runtime. Here, we can have up to one hundred client instances for one and only one server instance. The sets of instances are connected through a port typed by the Facet interface and supported by a connector. At this level, only one protocol must be chosen. This stage is called the Assemblage.

We model the infrastructure and/or hardware constraints in terms of computation node types and communication medium types. The set of instances are mapped to nodes and connectors are mapped to media. Nodes can be geographically located in a site. This stage is called the Deployment.

3.2 Modelling constructs

We detail here the concepts of the DAD language. The DAD metamodel is presented in Figure 2 as a UML class diagram. Attributes have been hidden to keep the diagram simple.

The interface definition mechanism is presented in Figure 2a. An Interface groups one or more Services, subtyped by Operations, Events and Exceptions. Operations are synchronous or asynchronous distant procedure calls with
Parameters and a possibly non-void return value. Events are formatted messages produced without being remotely called. Exceptions are fault messages produced during a synchronous Operation call and used to warn the caller that some error occurred. A Parameter is typed by an Interface, a PrimitiveType or a custom DataStructure.

The Definition layer constructs are depicted in Figure 2b. Interfaces are exposed in an architecture model as Facets. A Facet is used or implemented by a ComponentType. Compatible Facets are linked through a LinkType. A LinkType is either a ConnectorType or a DelegationType. A DelegationType can express point to point communications, broadcast-like communications and peer to peer-like communications. A DelegationType is used to connect Facets implemented into or used by an inner ComponentType to a parent ComponentType. LinkTypes accept a list of Protocols that are used to specify the communication protocols. Protocol expresses the properties and rules of the communication. LinkTypes define how objects are connected and Protocols focus on how they communicate. This separation between LinkTypes and Protocols offers the possibility to specify a wide range of connectors from procedure calls to web services, for example. As LinkTypes, a Facet restricts also the list of supported Protocols. ComponentTypes can also be connected by a usage relationship that says nothing about the involved Facets but declares a dependency between two ComponentTypes. This kind of relation offers more flexibility than a fully-typed Facet and can be useful when the nature of the interfaces is not yet known or undocumented.

Figure 2c. presents the Assemblage constructs. We incarnate each ComponentType as a SetOfInstances saying how many instances of a ComponentType may be present at runtime. A SetOfInstances can be managed by other SetsOfInstances, or can be identified as manually-started. This may be a useful piece of information for the deployment strategy. SetsOfInstances are connected by Links through Ports. A Link is typed by a LinkType and a Port is typed by a Facet. A specific Protocol must be chosen in the list of accepted Protocols. This level models the concrete system architecture with concrete instances that can appear and disappear during the IS lifetime.

We also define a set of infrastructure-related constructs as presented in Figure 2d. These modelling objects are used to create an abstract representation of the deployment infrastructure. A NodeType defines the machine type that will host part of the architecture. A MediumType defines a physical link type that can be used for the communication. The Deployment stage consists in mapping the architecture
model to the target infrastructure. Each SetOfInstances is mapped to a Node typed by a NodeType. Each Link is mapped to a Medium. A Medium is defined among others, by the supported and/or rejected Protocols. Nodes are possibly grouped in geographical Sites (mainly for documentation purpose).

3.3 Refinement of construct semantics

Properties can be defined for many modelling constructs. This attribute mechanism allows designers to refine the construct semantics with domain specific or hardware-related properties. These attributes have two goals: tagging a construct with some properties and validating deployment compatibilities. First, by annotating constructs with properties, we intend to raise the model documentation and clarity because important assets of constructs are directly shown on the model itself. Second, attributes can be added to Assembly constructs and hardware-related specifications to deployment constructs, in order to validate the feasibility of the specified quality attribute. Types can be refined by quality attributes concerning reliability, portability or efficiency as defined in the ISO/IEC 9126 standard [13]. Instances can be tagged with platform specific key-value attributes, such as implementation technology, programming language, hardware requirements and so forth. The infrastructure construct semantics can be refined by attributes in order to express deployment and hardware constraints such as available disk space, CPU power, link bandwidth, supported communication protocols, etc.

Some constructs can inherit from previously defined constructs of the same type. This is mainly the case for Protocols, ComponentTypes and Nodes. For example, a remote procedure call (RPC) Protocol can be later refined into a CORBA Protocol [25]. Another example is a web client ComponentType refined into a secured web client. As a last example, one can have a specific personal computer Node extended by another one with a larger hard disk drive.

As highlighted in [3], behavioural specification of Services and Protocols must be expressed in an abstract way for many purposes. Modellers write a minimal behavioural specification for a ComponentType and refine it for the corresponding SetOfInstances in order to enhance implementation flexibility. The specifications will be used by developers to know exactly the aim of a Service and will help for writing test cases. Also, a set of property checks can be performed on such representations instead of working on the code itself. Likewise, one can ensure that different implementations of the same construct type are acting in the same way, so conform to the same specifications.

4. ARCHITECTURALLY SIGNIFICANT REQUIREMENT MODELLING LANGUAGE

4.1 Overview

We created a simple graphical notation in order to capture and keep track of Architecturally Significant functional and non-functional Requirements (ASR), as well as their impacts on architecture models. We do not address requirement analysis and refinement phases, but provide a tool for design decision traceability expressed as model transformations. The Figure 3 shows a sample ASR model.

An Interface or Definition DAD construct is linked by a single line to its related functional or non-functional ASR. A refining ASR is bound with a full arrow and an alternative ASR with a dotted arrow. A Selected model transformation set (or delegated constructs) is linked to its implemented ASR using a line with crosses at each end. Alternative implementations are linked with dotted arrows.

![Figure 3: Architecturally Significant Requirement sample model](image)

4.2 Modelling constructs

The Figure 4 presents the metamodel of the ASR modelling language.

![Figure 4: Architecturally Significant Requirement metamodel](image)
into the architecture model. A TransformationSet is composed of TransformationRules involving ModelContracts. Other implementation alternatives can also be modelled for documentation purpose. All DAD construct types present in the Definition layer must be root of an ASR tree only once in an ASR model.

4.3 Writing guidelines for requirement descriptions

With such a basic modelling language, we have to restrict the range of requirements we focus on. We need to work on requirements that already have a concrete meaning for the system architecture. We reuse here a part of the specifications presented in [1].

The requirement descriptions should be as direct as possible, use very limited vocabulary and without ambiguities. The key words (here, mainly the Definition constructs) must be defined in a lexicon. Any requirement should focus on only one result. When hypotheses are taken into account for a requirement, they should be in the requirement description too.

For all descriptions, the responsible architectural construct must be clearly identified. For all interactions specifications (the Services), all involved resources (ComponentTypes and LinkTypes) must be clearly identifiable in the description. For functional requirements, the description should contain the parameters types, the return value, the possible exceptions raised and the calling type (if no result is given).

For example, a QoS requirement can be "The Server shall respond to Client requests in max 5 seconds." Another functional requirement example can be "The Client requests is composed by a string value. The Server returns a boolean value saying if the request has been correctly handled or not."

5. COMPONENT-BASED DESIGN METHOD

As stressed by [22, 5, 15], developing a complex distributed information system should be an iterative task in order to cope with its size and complexity. Moreover, as proposed in [14], an architecture design can be seen as a set of design decisions. Our approach proposes a transformation-oriented method going from a coarse-grained architecture model to a fine-grained model where each transformation step is related to one design decision. As a preliminary task to our method, designers have to identify architecturally significant requirements and coarse-grained architectural elements complying to our language constructs. We use here the requirements that have a meaning on the system architecture, such as needed and offered services by components, Quality of Service requirements, design prerequisites, infrastructure constraints, etc. A requirement is linked to one coarse-grained model construct that will be in charge of its implementation. A couple of existing techniques, slightly adapted to our modelling language, can be used for this initial task. We particularly note the Twin Peaks model [23], the method based on the architecture prescription language [6] and the CBSP approach [11]. Final version of requirements should be written conforming the guidelines we gave in Section 4.3.

The list of requirements is summarised in the model presented in Section 4 and the identified architectural constructs are used to create a first coarse-grained architecture as presented in Section 3. Using our design method, designers will iteratively select one requirement and either refine it or define a model transformation that finally implements the requirement. This way, any change in the architecture model will be justified by a model transformation. Furthermore, as any modification in the system design focuses on an identifiable part of the architecture, we intend to highlight possible architectural inconsistencies more accurately, such as Quality of Service property violations, deployment problems, and so forth.

6. AN EXAMPLE: DISTRIBUTED IMAGE RENDERER

We present here an illustration of our design method based on an academic example. We will build a distributed and robust version of an image rendering software, called Pov-Ray [28]. Briefly, this freeware allows users to generate high-quality pictures from text-based description files using the ray tracing principle.

6.1 Preliminary task

As a preliminary task, we identified four ComponentTypes: a Client, a TaskDispatcher, a WorkerSlave and the COTS Pov-Ray. The Worker Slave will use Pov-Ray. The Client will contact the Task Dispatcher over the Internet. The Task Dispatchers and Worker Slaves will communicate over a Virtual Private Network.

Also, we formalised the architecturally-significant requirements following the method guidelines presented in Section 4.3. We list first the functional requirements.

1. The Client will provide a GUI-based application to allow users to submit a rendering job to Task Dispatcher.
2. The submission of a rendering job consists in a client reference, a description file (of type .pov) and an image width and height (in pixel). As an acknowledgement, the Client receives a job ID. The result of a rendering job is a .png image. The Client can disconnect after having submitted a job.
3. The Client can check periodically on the Task Dispatcher for its job results using the job ID.
4. The submission of a rendering sub-job by the Task Dispatcher to Worker Slaves contains the same info as a job completed with the starting and ending coordinates (in pixel). The submission is asynchronous.

Then, here is the description list of the non-functional requirements. We assume here that for some non-functional requirements, we already refined them by functional requirements meant to implement the parent requirement.

1. The Task Dispatcher robustness will be implemented by the replication of the ComponentType.
   (a) the coordination between Task Dispatcher will be implemented by an election algorithm.
   (b) a synchronisation mechanism between Task Dispatcher must be implemented to replicate submitted jobs on every Task Dispatcher.
2. Since the Worker Slave are unreliable resources, the Task Dispatcher will implement a job recovery mechanism.
(a) a pinging mechanism must be implemented to check if a busy Worker Slave is still alive and connected.

(b) a resubmitting mechanism must be implemented by the Task Dispatcher if a Worker Slave did not send its generated image before leaving/crashing.

6.2 First models and hypotheses

The Figure 5 shows the first DAD model with all identified constructs. We have the four ComponentTypes, their four related SetsOfInstances and three Nodes. The Client and Task Dispatcher are connected over the Internet, and the Task Dispatcher and the Worker Slave are in a virtual private network. As the interface with Pov-Ray is not defined yet, a usage dependency links it to the Worker Slave. This expresses an unsatisfied dependency that must be clarified by later design decisions.

For the ConnectionTypes, a set of supported communication protocols must be defined: the Internet ConnectorType supports TCP/IP [7] and the VPN ConnectionTypes supports VPLS [16, 19]. Note that we do not consider every possible protocols in order to stay simple and clear.

We define the cardinalities of the SetsOfInstances and we map them to the abstract infrastructure. For all c SetofInstance, a cNode is created. The same rule applies to ws. We decide to deploy the Pov-Ray COTS on the same Node as the Worker Slave. For readability reason, we detail the infrastructure specifications in sub-section 6.5. Nodes (resp. Media) are also typed by NodeTypes (resp. MediumTypes) where hardware-related constraints are expressed. As physical medium, we suppose that nodes are linked with CAT-5E [31] ethernet cables.

Note that all model constructs are not fully detailed on the graphical representation. As examples, we show the DoJob Interface and the TCP/IP protocol in Listing 1 in textual syntax.

```java
interface DoJob {
    sync boolean renderImage (in string clientID , in byte[] file , in integer height , in integer width );
    accepts TCP/IP;
}
```

Listing 1: Excerpt of the first DAD model in textual syntax

The Figure 6 shows the first ASR model summarising the requirements and their targets.

6.3 Functional alternatives

Following our proposal, we will select a requirement and either refine it or create a set of transformations rules that weaves the requirement into the architecture model. For a couple of requirements, a list of implementation alternatives can be represented in the ASR model. For the Job synchronisation requirement, we identified two alternatives: a full or a partial synchronisation. Briefly, the Partial synchronisation will only synchronise between the TaskDispatchers the information about the submitted jobs. The Full synchronisation will also synchronise the data related to the sub-jobs already completed by WorkerSlaves. Figure 7 illustrates the functional alternatives linked to their implementing interfaces. Let’s decide we choose the Full synchronisation solution.

We also have to define a set of transformation rules that will inject the newly defined interfaces into the architecture model. Informally, we need to execute the following transformations:

1. create one new Interface: FullSync and add it in the implementation and usage list of the TaskDispatcher
2. constraint the connection between the TaskDispatchers with the VPN ConnectorType
3. define new cardinalities for the td SetofInstances (since the TaskDispatcher will be replicated now)
4. create one Port typed by the new Interface
5. create one vpls Connector typed by the VPN ConnectorType
6. create a connection between tdNodes with a cat5e Medium.
7. map the new vpls Connector on the new cat5e Medium.

Figure 8 illustrates the modified architecture model.

6.4 Implementation possibilities

For each interface, a set of non-functional requirements can alter the functional semantics. Like ComponentTypes, Interfaces must be the root elements of a decision tree. Let’s say that we may want to perform the operations offered by the DoJob interface on a secured or unsecured connection. We can now specify more precisely the DoJob interface with new operations (connect()) and the accepted protocols that satisfy the requirements. In Figure 9, we selected the secured connection alternative.

6.5 Infrastructure constraints impacts

Infrastructure constraints can influence the final design of an architecture. When such constraints are taken into account lately in the system design, this can result in incompatibilities between the software and the supporting hard-
ware.

Also, as highlighted in Peter Deutsch’s eight fallacies of distributed computing, the network topology can be subject to changes during the software lifetime. Let’s suppose that the connection between the Client and the Task Dispatcher goes through a firewall. This firewall has strict routing rules and does not allow RPC communication. The new deployment part of the modified DAD model is presented in Figure 10. We only introduce here the firewall Node in the middle of the communication between the Client and the Task Dispatcher. No resolution about the new constraints is taken into account at this point.

Figure 10: DAD deployment excerpt with the firewall

As explained in Sub-section 3.2, a Medium is also constrained by a set of accepted or rejected Protocols. The firewall is represented as a Node that can restrict the Protocols accepted by the Media linked to it. It does not host any SetOfInstances since no part of the architecture must be deployed on it. A new MediumType (CAT-5E-NoRPC) must be introduced to link the firewall to the cNode. We reject all RPC-based protocols for this new MediumType (remember that we have inheritance for Protocols). The instantiated Medium has to host the rpc Connectors, so that the incompatibility is clearly shown on the model. We also introduce a Site in order to geographically group the firewall and the tdNode for readability and documentation purposes. Note that the accepted and rejected protocols are not represented in the graphical model for readability reasons.

All these constraints and decisions must be reflected in the ASR model. Figure 11 illustrates these modifications. Since we already decided to use a secured connection, we only show the decisions for that alternative. Also, we only provide the model for the DoJob Interface. The same reasoning applies to the other Interface offered to the Client.

In order to integrate the new constraint in the architecture, we identified two alternatives: (i) modify the Task Dispatcher implemented Interface and use another Protocol; (ii) use a wrapper that handles requests from Clients and forward them to the Task Dispatcher. We decide to go for the second solution, so we create a new ComponentType in charge of wrapping the requests from one Protocol to another.

Figure 11: Communication protocol constraints for the DoJob Interface

7. DISCUSSION

The explored alternatives are tracked in the requirement model with related transformations sets (if existing). This way, we keep track of the discarded alternative and we can retrieve them later on if the selected solution created model conflicts in a later model transformation.

The documentation and design decisions traceability is enhanced because any new ASR is linked to refinements or alternatives and is also linked to the model transformation that implements it. Any architecturally-significant requirement is weaved one-by-one into the architecture model, so that any change from the first coarse-grained architecture model is documented.

In case of having a set of COTS conforming to the same interface specification, all these alternatives can be tracked in the ASR model with their corresponding constraints (communication protocol, calculation/space constraints, etc).

By separating communication channel and protocol, we intend to offer more flexibility for the communication specification. The concrete communication between components must conform to a protocol (for example, a Remote Procedure Call, an home-made TCP/IP-based protocol or a Java method call), many possibilities can be explored without influencing much of the model.

Property violations can be detected directly as they appear in the model since the links between the Definition, Assemblage and Deployment levels are maintained for all transformations. For example, if a designer wants to replace a LinkType by a new one with a minimum bandwidth requirement that the underlying Medium cannot guarantee, a corrective decision will be needed before going further in the development process.
8. LIMITATIONS

Our approach requires a non-trivial preliminary step in order to identify architecturally significant requirements and coarse-grained components. Also, during the development life-cycle, no decision support mechanism is proposed, neither a requirement decomposition or refinement method. Our proposal offers traceability and documentation facilities by hardly coupling requirements implementations and related model transformations.

At present time, only a small set of association types between ASR are taken into account. The language should integrate relations such as conflicts or positive and negative impacts. This point still needs further investigations.

Our approach on variability over design decisions can be easily extended on architecturally significant requirements themselves. Modellers can define alternative ASR and select only the ones they need on a particular architecture model.

Even if we plan to make model transformations as generic as possible, software engineers will have to define their own transformation rules when they encounter specific needs. We have to provide a proper transformation mechanism in order to enhance the re-usability of transformation rules.

9. FUTURE WORK

The attribute mechanism presented in Section 3.3 has to be developed. We intend to create an extensible taxonomy of attributes and let designers free to define their own attributes and associated validation rules.

We intend to investigate behavioural specification facilities. As suggested in [3], distant services should be abstractly specified. A set of labelled transition system languages exist such as Communication Sequential Processes (CSP) [12], Finite State Processes (FSP) [20] and UML state charts [27]. Adding such features to services and communication protocols has a triple goal. First, it helps software engineers to have a clear understanding of the semantics of these model elements. Second, it can be used for partial validation of components composition. Third, it can be used for system animation purpose.

We have to develop the transformation modelling facilities. One major challenge for such languages is the ability for designers to write reusable transformations in order to incrementally build a set of automated transformations for recurrent problems. Another challenge is the ability to restart from any intermediate model, build or select another available transformation alternative and reapply the remaining transformations into the new model.

10. CONCLUSION

In this paper, we presented a transformation-oriented design method for distributed systems. Our approach is based on an architecture description language (ADL) and a requirement modelling language intended to enhance evolution and variability capabilities of system architectures. We explained the basic concepts of the ADL and the three-layer representation of architecture models. We presented our requirement modelling language focusing on architecturally significant functional and non-functional requirements. We have to provide a proper transformation mechanism in order to enhance the re-usability of transformation rules.

11. ACKNOWLEDGEMENTS

The authors thank Patrick Heymans, Martin Mahaux and Nicolas Genon (all from PReCISE) for their contributions, reviews and advices.

12. REFERENCES

