Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Architectural Rules Conformance - with ArCon and Open-Source Modeling Tools

by

Emil Fridell

LIU-IDA/LITH-EX-A—12/031—SE

2012-06-12

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköping University
Department of Computer and Information Science

Final Thesis

Architectural Rules Conformance – with ArCon and Open-Source Modeling Tools

by

Emil Fridell

LIU-IDA/LITH-EX-A—12/031—SE

2012-06-12

Supervisor: Olena Rogovchenko (IDA)
Gert Johansson (Combitech)

Examiner: Peter Fritzson

Abstract

In software development it is often crucial that the system implementation follows the architecture defined through design patterns and a constraint set. In Model-Driven development most artefacts are created using models, but the architectural design rules is one area where no standard to model the rules exists. ArCon, Architecture Conformance Checker, is a tool to check conformance of architectural design rules on a system model, defined in UML, that implements the system or application. The architectural design rules are defined in a UML model but with a specific meaning, different from standard UML, proposed by the authors of ArCon. Within this thesis ArCon was extended to be able to check models created by the Open-Source modeling tool Papyrus, and integrated as a plugin on the Eclipse platform. The method used by ArCon, to define architectural rules, was also given a short evaluation during the project to get a hint of its potential and future use. The case-study showed some problems and potential improvements of the used implementation of ArCon and its supported method.

Contents

List of Figures

Chapter 1

Introduction

This chapter is the introduction to a master thesis project (30 credit points) final report examined at the Department of Computer and Information Science (IDA) at Linköping University. The thesis is the final part of a 5 year degree leading to a Master of Computer Science and Engineering. The thesis work has been performed at Combitech AB in Linköping. Combitech AB[1] is a software consultancy company with expertise in information security, systems integration, communications, mechanics, systems security, systems development and logistics.

1.1 Background

In software development there is often an architectural model defined by the architect which describes the system in a high level with rules that the realizing system model must full-fill. ArCon (Architectural Conformance Checker) is a tool that checks a Unified Modeling Language (UML) model against defined rules in another UML model. The ArCon tool can be used to automate the validation between the architectural rules and the system model. The rules are modelled in a subset of UML where the UML constructs have been rendered a different meaning than what is prescribed by the standard.

ArCon can read out the architectural rules, and then based on these review a system/software-model and identify deviations from the rules. This can save a large amount of manual review work and also assist in the transfer of architectural knowledge to the developer through the architectural model and by means of an automatic audit that can be applied more or less continuously during the work.

[1] For more information about Combitech AB, please visit www.combitech.se

1.2 Purpose and Objective

The first version of ArCon can only read models from the Rational Rhapsody modeling tool by connecting to Rhapsody's application interface (API). The industry has requested more Open-Source tools and specifically the ITEA2-project OPEES, which ArCon has connections to, has focus on supporting Open-Source tools for software development. ArCon itself is Open-Source and by extending ArCon, to support Open-Source modeling tools, would fully support its objective on the Open-Source scene.

The Papyrus modeling tool is an Open-Source modeling tool, and it supports the UML standard. Thereby Papyrus was selected, by Combitech, as the targeted modeling tool to by supported by ArCon.

1.3 Limitations

Because of the limited time within a thesis project, the main focus was to support only one Open-Source modeling tool. To make future support for other modeling tools easier, the solution to support the Open-Source tool was to be made as generic as possible. The best case scenario was identified as a solution that enables support for other modeling tools even if that was not a direct requirement, that would be tested or validated.

1.4 Intended Audience

This report is intended for audience familiar with software development, and having at least basic knowledge of UML. A simple introduction to UML is given in Chapter 2 on page 5 to give readers unfamiliar with UML the basic concept behind it. Readers that are familiar with the concept, but feel that they need to refresh their memory, may also find the section valuable to read. There are also a few different programs that are referred to in this report which may not be known to all readers. These will also be explained briefly in the Technologies chapter.

1.5 Thesis Overview

This section will explain how this report is constructed and give a basic description of the content presented in each chapter. The purpose of this section is to let the reader get a quick overview so that he/she can decide what to read.

- **Chapter 2** gives a basic description of software and standards that this report comes in contact with. Its only purpose is to give a basic description for readers unfamiliar with these concepts.

- **Chapter 3** describes how architecture conformance is checked today, why it is checked, some existing problems and how this motivates the development of ArCon. The method to describe architectural design rules, proposed by the authors of ArCon, is presented at the end of the chapter.
- **Chapter 4** deals with the development and extension of ArCon which was the aim of this thesis project.
- **Chapter 5** is a short user guide on how to install and use ArCon in the Eclipse environment. The chapter also contains some simple models with the purpose to show examples how ArCon can be used.
- **Chapter 6** goes over the case study performed with ArCon on an industrial project.
- **Chapter 7** summarizes the thesis, and gives suggestions on the future work and development of ArCon and its related method to define architectural design rules.

Chapter 2

Technologies

This chapter will present the different technologies applied in this thesis. The purpose is to give a basic description of the chosen systems and standards to readers who are not familiar with them. If the reader is already familiar with a system or standard described in this chapter, then the section may be skipped entirely.

2.1 UML

This section will give a basic informal description of what Unified Modeling Language (UML) is, see homepage[1] or community forum[2] for specific details and answers.

UML is a standardized general-purpose modeling language in the field of object-oriented software engineering. The standard is managed, and was created, by the Object Management Group (OMG). It was first added to the list of OMG adopted technologies in 1997, and has since become a widely used and recognized industry standard for modeling software-intensive systems. [12]

A basic informal overview[3] of UML is: ,,*The Unified Modeling Language (UML) is used to specify, visualize, modify, construct and document the artifacts of an object-oriented software-intensive system under development. UML offers a standard way to visualize a system's architectural blueprints, including elements such as: activities, actors, business processes, database schemas, (logical) components, programming language, statements and reusable software components.*

UML combines techniques from data modeling (entity relationship diagrams), business modeling (work flows), object modeling, and component

[1] http://www.uml.org/, April 2012
[2] http://www.uml-forum.com/FAQ.htm, April 2012
[3] http://en.wikipedia.org/wiki/Unified_Modeling_Language, April 2012

modeling. It can be used with all processes, throughout the software development life cycle, and across different implementation technologies. UML has synthesized the notations of the Booch method, the Object-modeling technique (OMT) and Object-oriented software engineering (OOSE) by fusing them into a single, common and widely usable modeling language. UML aims to be a standard modeling language which can model concurrent and distributed systems. UML is a de facto industry standard, and is evolving under the auspices of the Object Management Group (OMG).

UML models may be automatically transformed to other representations (e.g. Java) by means of QVT-like transformation languages. UML is extensible, with two mechanisms for customization: profiles and stereotypes." An example of an UML class diagram can be seen in Figure2.1.

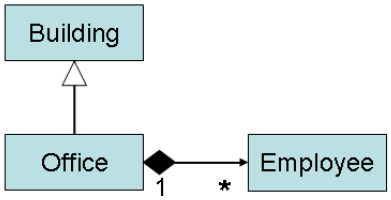


Figure 2.1: Simple example of a UML class diagram

2.2 ArCon

This section will explain what ArCon (Architecture Conformance Checker) is.

It was first developed by Anders Mattsson at Combitech AB to be used as proof of concept for their proposed approach to model architectural design rules and check the conformance of these rules against a system realization model (called system model) with tools. ArCon is the tool, designed to be used, to automate the validation between the architectural rules and the system model. The rules are modelled in a subset of UML where the UML constructs have been rendered a different meaning than what is prescribed by the standard. (More information about ArCon's architectural rules syntax is provided later in 3.3.2 on page 14)

ArCon can read out the architectural rules, and then based on these review a system/software-model and identify deviations from the rules, and report these deviations. The idea is that it can save a large amount of manual review work and also assisting in the transfer of architectural knowledge to the developer through the architectural model and by means of automatic audit that can be applied more or less continuously during the work.

The first version of ArCon could read and check UML models only from

projects defined in the Rational Rhapsody modeling tool (see Section 2.3 on page 7 for more info about Rhapsody).

ArCon is an Open-Source project which is available, at the time of writing, at http://code.google.com/a/eclipselabs.org/p/arcon/[4].

2.3 Rhapsody

Rational Rhapsody, also referred as Rhapsody in this report, is known as IBM Rational Rhapsody after it became an IBM Rational product following the acquisition of Telelogic AB in 2008. Rhapsody[5], a modeling environment based on UML, is a visual development environment for systems engineers and software developers creating real-time or embedded systems and software. Rational Rhapsody uses graphical models to generate software applications in various languages including C, C++, Ada, Java and C#.

Rational Rhapsody aims to help diverse teams collaborate to understand and elaborate requirements, abstract complexity visually using industry standard languages (UML, SysML, AUTOSAR, DoDAF, MODAF, UPDM), validate functionality early in development, and automate delivery of high quality products.

2.4 Eclipse and Papyrus

Eclipse[6] is an open source community, whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle. The Eclipse Foundation is a not-for-profit, member supported corporation that hosts the Eclipse projects and helps cultivate both an open source community and an ecosystem of complementary products and services. One of the more recognized projects within the Eclipse Foundation when talking about Eclipse is the Eclipse IDE (integrated development environment) which offers a large modifiability with extensions in the form of plugins.

Papyrus is one of these extensions for the Eclipse IDE. Papyrus[7] is a component of the Model Development Tools (MDT) subproject aiming at providing an integrated, user-consumable environment for editing any kind of EMF model and particularly supporting UML and related modeling languages such as SysML and MARTE. Papyrus provides diagram editors for EMF-based modeling languages amongst them UML 2 and SysML and the glue required for integrating these editors (GMF-based or not) with other

[4] http://code.google.com/a/eclipselabs.org/p/arcon/, May 2012
[5] http://www-01.ibm.com/software/awdtools/rhapsody/, April 2012
[6] http://www.eclipse.org/org/, April 2012
[7] http://www.eclipse.org/modeling/mdt/papyrus/, April 2012

MBD and MDSD tools. It also offers a very advanced support of UML profiles that enables users to define editors for DSLs based on the UML 2 standard and its extension mechanisms. The main feature of Papyrus regarding this latter point is a set of very powerful customization mechanisms which can be leveraged to create user-defined Papyrus perspectives and give it the same look and feel as a native DSL editor.

Chapter 3

Software Architecture

This chapter will explain why software architecture and specially architecture rules are used today, why this is a topic that is of interest for this thesis and lastly how the proposed method to define architectural design rules, which ArCon checks, is designed.

3.1 Motivation

Software Architecture allows for early assessment of and design for quality attributes of a software system, and it plays a critical role in current software development [13]. A primary role of the architecture is to capture the architectural design decisions [5], and its derived design rules. In many projects within Model-Driven Development (MDD), UML models are the first artefacts to systematically represent a software architecture [7]. One of these important artefacts of the software architecture is the architecture rules. In this report, the meaning of architectural design rules is rules (including constraints), defined by the architect, to be followed in the detailed design of a system. The architectural design rules are typically an abstraction of the software design with no or very little detail specification on the implemented system [10]. To make sure the system follows the design decisions and rules, verification and validation of the rules against the implemented system model becomes important.

3.2 Drawbacks and Problems Today

In a recent study of UML based software architecture and design some problems in the industry were recognized [7]. One of these important issues the researchers found and requested improvement on was:

- More consistency between UML models and system requirements as

well as implementations. Better mechanisms for traceability and round-trip engineering were suggested to help reducing these problems.

Anders Mattsson, Björn Lundell and Brian Lings also came across this problem, in their research [9], with the inability to capture the architecture rules in an efficient way. In the article they stated there are very few works on a standard way to define architecture design rules, and no real definition on how to do it. During their research they found that the inability to formalize the design rules led to need for manual enforcement of the rules on the system. They also noted that such manual enforcement was both an error-prone and time-consuming task that took a big part of the architects' time during the development. All parts in the system had to be reviewed to make sure they followed the architectural rules before they could proceed into full implementation. As a result the architects became bottlenecks in the projects because the different parts were waiting for an architectural review. The effect of this was a number of problems such as:

1. **Stalled detailed design:** The design teams have to wait for the architects to review their overall design before they can dig deeper into the design.
2. **Premature detailed design:** Design teams start detailing their design before their overall design is approved by the architect, with the risk that they will have to redo much work after the review.
3. **Low review quality:** Low quality of the reviews, leading to problems later in the project.
4. **Poor communication of architecture:** The architects have no time to handle the communication with the design teams regarding architectural interpretations or problems, problems are "swept under the carpet."

The same authors later stated, in another article, that "The state of the art is to capture these rules in informal text" [10]. By 'these rules' they were referring to the architectural design rules. They also noted that using informal text becomes a problem in MDD since MDD relies on models to increase development efficiency through automation. The authors' goal was to find a method to describe the architecture using models, just like other artefacts in MDD, and to be able to check the conformance with tools so that the manual checking would be reduced.

Other researchers states that most works on software architectures focus on designing architectures carefully to avoid change, as changing the architecture after the initial design is costly [4, 5]. Some of the reasons they pointed out were:

- **Design rules and constraints are violated**: During the evolution of the system, designers can easily violate design rules and constraints

arising from previously taken design decisions. Violations of these rules and constraints lead to architectural drift and its associated problems (e.g. increased maintenance costs). As design rules and constraints influence future design decisions, they have a steering influence on the future direction of the architecture.

- **Obsolete design decisions are not removed**: When obsolete design decisions are not removed, the system has the tendency to erode more rapidly. In the current design practice removing design decisions is avoided, because of the effort needed, and the unexpected effects this removing can have on the system.

The thoughts behind this also validate the other researchers request to be able to define, edit and validate the architecture rules in an easy way. It has also been said that the architecture of a software system is a critical artefact in the software life-cycle and should be evaluated as early as possible [8, 4]. The architecture should also be checked continuously during the development to ensure the system follows the architectural design at all important milestones. Given the methods currently available and the related issues, while architecture is recognized as important, the authors were led to believe that better tools to define, manage and verify the architectural rules are needed.

3.3 Using UML to Define Architectural Rules

Anders Mattsson, Björn Lundell, Brian Fitzgerald and Brian Lings later proposed one way to address the problems, described in 3.2, by defining a way to use UML modeling to describe the architecture rules [10, 11]. Modeling the rules in UML gives a few advantages according to the authors:

- **Similarity**
  Both the architecture rules and the system implementation can be modelled using UML. This allows both models to be modelled using the same tools that both the architect and developers are familiar with.

- **Rule Validation**
  Tools can validate and check the architecture rules which reduce the eruptions and enhance the quality of the design. Changing the model becomes easier and less expensive because less time is needed to check and validate the new design and rules.

- **Consistency and validation**
  Using a formal way to define the rules like UML not only allow tools to check and validate the design of the architecture, but also allowing tools to be developed to automatically check and validate the rules against the system implementation. This reduces the manual work

that would have been needed to make sure the rules and the implemented system maintain consistent.

3.3.1 Building Blocks for Compliance Checking

The authors of the article [1] stated that the required tool support for architectural compliance checks must consist of the following basic building blocks:

1. A formal notation to represent the intended architecture or reference architecture.
2. A formal notation to represent the design or implementation of the application.
3. A formal notation to represent mappings between intended architecture and design/implementation.
4. A compliance checking technique based on there presentations of architecture, design, and mapping.

The proposed method that ArCon supports contains all those blocks and can be recognized as:

1. Using a modified UML syntax the intended architecture is described in UML using the transformation rules described in Section 3.3.2.
2. A standard UML model is the system model that represents the design or implementation of the application.
3. The stereotypes transformed from the architecture model is used on the system model and acts as the mapping between the intended architecture and the design/implementation.
4. Using the above blocks ArCon can read both models and check the compliance between them. The transformation rules results in regulations that the architecture model builds rules and constraints on, and ArCon checks if the elements within the system model follows these rules.

Artefacts All four blocks can be seen as artefacts within the proposed solution, where the fourth artefact is ArCon itself and its regulations. The other three artefacts are divided into two main artefacts and one sub-artefact. The main artefacts are the architectural rules model and the system model. The architectural rules model defines the rules which the system model must follow. This architectural model is defined in UML using a different syntax than the standard, later described in Section 3.3.2. The system model is the software's realization, and it is also defined in UML but it follows the standard UML specification. Following the standard is a

must since it is the implementation model which the software product is to be created from. The sub-artefact is an UML profile, in this report called architecture rules profile, which is applied on the system model. The constructs in the architectural rules model transforms into stereotypes to be defined in the profile. The system model uses the stereotypes defined in the profile, and all classes using the stereotypes must follow the rules defined for those stereotypes. The interactions between the three artefacts are visualized in Figure 3.1. For more information on how the transformations are defined, see Section 3.3.2.

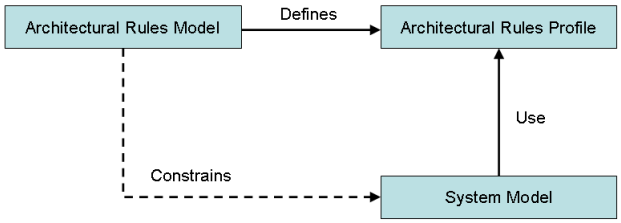


Figure 3.1: Overview of the artefact interactions

An advantage with the method compared to other methods in the same area, like reflexion models, is that the mapping between the models is more automatic or indirect. When extracting the reflexion model from a system implementation the mapping becomes a semi-automatic or extra task, which the architect has to manually perform by defining the mapping between the intended architecture and the reflexion model. The authors of the article *Architecture Compliance Checking - Experiences from Successful Technology Transfer to Industry* give a basic description of reflexion models and how the mapping is performed [6]:

"*Reflexion models compare two models of a software system against each other: typically, an architectural model (the planned or intended architecture) and a source code model (the actual or implemented architecture). The comparison requires a mapping between the two models to be compared, which is a human-based task.*"

Using stereotypes the developers will add the stereotypes on the system model during the development, and no extra mapping has to be done afterwards. Profiles and stereotypes are UML's lightweight mechanism to extend the language without having to change the constructs of UML [3]. Thereby, applying stereotypes and use them during development is a natural method when developing with UML models. In many projects stereotypes are already used in the model development, and they can also be used by the architecture rules. As a result adding the architecture stereotypes offers little extra work, if any at all.

Automating Profile Creation With current tools both the architectural rules model and profile have to be created manually, but since the profile reflects the constructs defined in the model it is possible to automate the creation of the profile and its stereotypes. A possible scenario is that a modeling tool, which supports ArCon's way to define architectural design rules, would also have the functionality to generate the architectural rules profile based on the architectural rules model. This would remove the unnecessary work to manually create the profile.

3.3.2 Rules Translation

In this section the translation rules for defining architectural rules, according to the proposed method in Section 3.3, are presented.

In Table 1 the definition on how to interpret the constructs in the architectural rules model as constraints on a system model is given. The definitions refer to an architectural rules model complying with the form of the generic model given in 3.2. References to terms defined in the generic model are written in italics.

ArCon has some derivations from the transformations described in Table 1. The transformation affected is marked to be "Not yet supported by ArCon" or "Partially supported by ArCon". Transformation T6 is set to be partially supported by ArCon in the extent that it can only do a specific type of constraint. This report will not go into details how this transformation is partially supported, for detailed information see ArCon's User Guide[1].

One extra rule, which is not described in Table 1, has been added:

- **T0** The system model must be of stereotype `<<System_Model>>`.

Rule T0 makes sure the architectural rules model can always refer to, and constrain, the system model by creating a stereotype named `<<System_Model>>`.

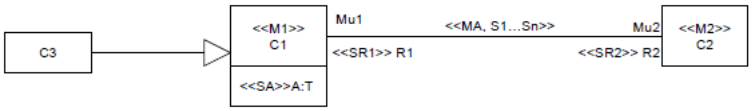
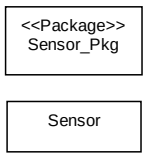
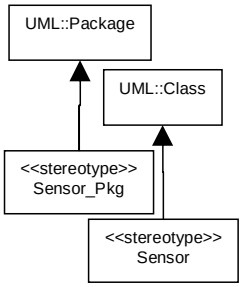
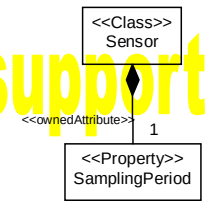
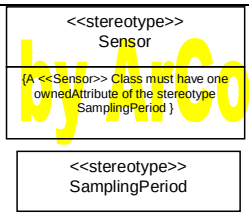
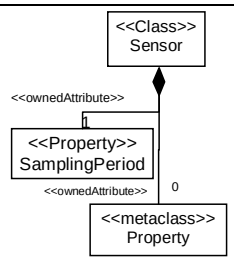
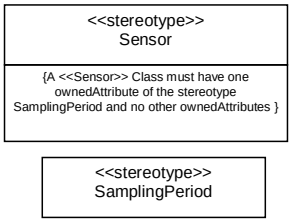
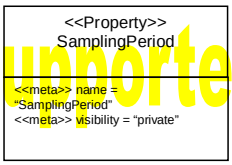
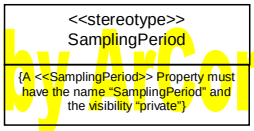
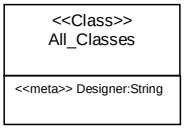
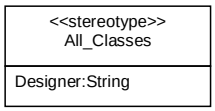
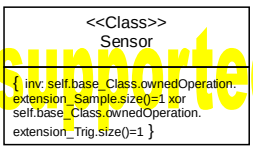
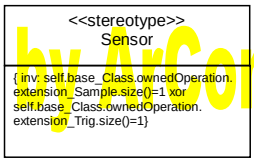
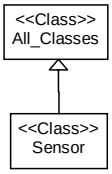
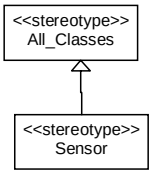
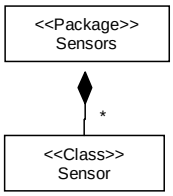
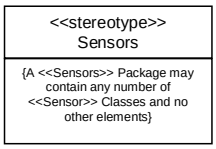
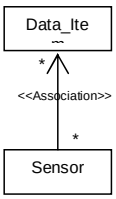
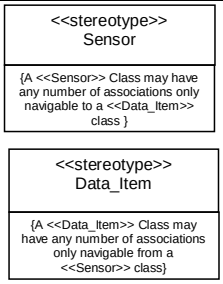
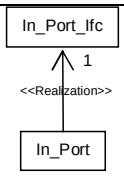
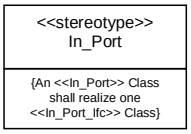
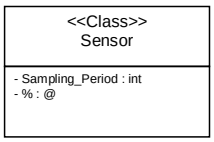
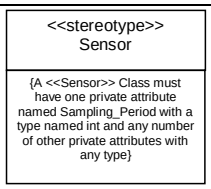


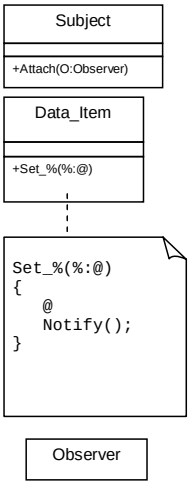
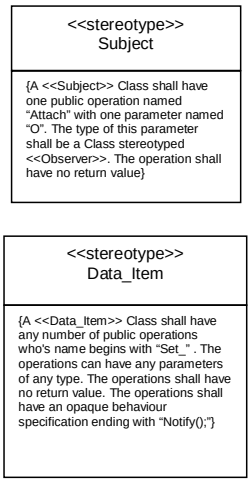
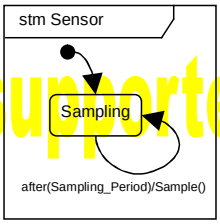
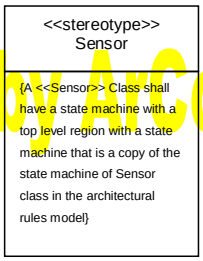
Figure 3.2: A generic architectural rules model used in the definition of transformations

[1] http://arcon.eclipselabs.org.codespot.com/files/Users_Manual.pdf, May 2012

Table 1. Definition of transformations between constructs in the architectural rules model and constraints on stereotypes in the system model

Transformation	**Example**	
	Architectural rules model	**System model stereotypes**
T1: A class named *C1* with the stereotype *M1* is transformed into a stereotype named *C1* extending the meta-class *M1* unless transformation number T3 below applies. If *M1* is undefined then "Class" is assumed		
T2: If *SR2* is the role in the UML meta-model on the far end of an association from the meta-class of *C1* to the meta-class of *C2* then the multiplicity of *R2* for a *<<C2>>* element shall be constrained to *Mu2* in stereotype *<<C1>>*		
T3: If *M1* equals "metaclass" then *C1* represents the class *C1* in the UML meta-model and is not transformed into anything in the system model. This can be used to specify constraints in other stereotypes in respect to these meta-classes		
T4: If *SA* equals "meta" far an attribute *A* and the name of *A* matches the name of an attribute of class *M1* in the meta-model then it is transformed into a constraint on that attribute on allowed values. The value of the attribute is constrained to match a regular expression specified as the default value of the attribute.		
T5: If no match is found for an *A* where SA equals "meta" (according to T4) then *A* is transformed into an attribute A of the stereotype (tag-definition), thus defining a tagged value to be set in the model elements where the stereotype is applied		
T6: Any OCL constraint in the context of a class *C1* is copied exactly as it is into the stereotype *C1*		

Transformation	Example	
	Architectural rules model	**System model stereotypes**
T7: A generalisation relationship from a class *C3* to a class *C1* in the architectural rules model is transformed to a generalisation from stereotype <<C3>> to stereotype <<C1>>. The UML meaning of this is that all constraints and attributes of the stereotype <<C1>> are inherited by the stereotype <<C3>> and that any <<C3>> element is also an <<C1>> element.		
T8: If M1 equals “Package” and the aggregation of R1 is “composite”: A <<C1>> Package is constrained to have Mu2 number of <<C2>> elements as packagedElements. The visibility of these elements shall be the visibility of Mu2. Also, a <<C1>> package is not allowed to have any packagedElements unless explicitly allowed in the model.		
T9: <<*C1*>> elements are only allowed to have the associations, dependencies, generalizations and realizations explicitly allowed according to T10 and T11.	See examples for T10 and T11.	
T10: If *MA* equals “Association”: A <<*C1*>> element shall be associated with *Mu2* number of <<*C2*>> elements. The association ends shall have the same navigability, aggregation (none, shared or composite) and visibility as *R1* and *R2*. The association ends shall also have qualifiers according to the qualifiers of *R1* and *R2*. The name and type of these shall be according to the transformations for attributes specified in T12. The association shall have the stereotypes *S1* to *Sn*.		
T11: If *MA* equals “Dependency”, “Generalization” or “Realization” and the association is only navigable from *C1* to *C2*: A <<*C1*>> element shall have a relationship according to *MA* to *Mu2* number of <<*C2*>> elements with stereotypes *S1* to *Sn*		
T12: If there are attributes *A* of *C1* where SA is not equal to “meta”: – All parts of the definition of an attribute of a <<*C1*>> class must match the corresponding part of an *A*, where the wild card characters “@” and “%” in any part of the definition of *A* can be replaced with any character sequence. Parts of *A* not specified (as for instance default value for Sampling_Period in the example to the right) – All *A* must be matched by one attribute in a <<*C1*>> class. An exception to this is if the name of *A* contains the wild card character “%”; in this case any number of matches (including		

Transformation	Example	
	Architectural rules model	**System model stereotypes**
zero) is allowed. – If the name of a type of *A* is identical to the name of a class C in the architectural rules model then the type of a matching attribute must be a <<C>> element.		
T13: If there are operations O of *C1*: – All parts of the definition of an operation of a <<*C1*>> class must match the corresponding part of an O, where, for each part of the definition, the wild card characters "@" and "%" can be replaced with any character sequence. Properties of O not specified (as for instance parameter directions for operations in the example to the right) are unconstrained. – This requirement holds for all parts of the definition of O defined in the UML meta-model, such as for instance opaque behaviour specified for the operation. – The character "%" in a parameter name means that the definition of this parameter can be repeated any number of times, including zero. In these parameter definitions "%" can be replaced with any character sequence. – If the name of the type of O or a parameter of O is identical to the name of a class B in the architectural rules model then the type of matching operations or parameters in the <<*C1*>> class must be of a <<B>> Class. – All O must be matched by one operation in a <<*C1*>> class. An exception to this is if the name of O contains the wild card character "%"; in this case any number of matches (including zero) is allowed.		
T14: If *C1* has a state machine then a <<*C1*>> class must have a state machine where there for each region in *C1* shall be an identical region in the <<*C1*>> class. The wild card character "@" may be used in the transition definitions in *C1* and shall then be matched with any text string in the corresponding transition in the state machine of a <<*C1*>> class. It is allowed to have additional regions in the state machine of a <<*C1*>> class.		

Chapter 4

ArCon Extension

This chapter explains the thoughts and process of the work to extend ArCon with support for Open-Source modeling tools.

4.1 Motivation and Requirements

This section describes the motivation and requirements for extending ArCon in this thesis project. It also explains few core decisions and design principles.

4.1.1 Open-Source Tool Support

The ITEA2-project OPEES works with tools that have a long life-cycle compared to many other tools (15-20 years). The OPEES homepage[1] describes their objectives as ,,*The mission statement of OPEES is "to settle a community and build the necessary means and enablers to ensure long-term availability of innovative engineering technologies in the domain of dependable / critical software-intensive embedded systems".*

For OPEES partners and supporters, this challenge can be achieved if we succeed in building an ecosystem in the open source frame, with the relevant business models, in order to ensure this long term availability of engineering tools and components."

For these kinds of tools there is a requirement to support the tools during the entire life-cycle, even if new operating systems or hardware emerge. One big risk is that the vendor of the tool disappears after a while or that the tool simply loses further support from the vendor. To address this requirement and handle the risks the OPEES project has a focus on Open-Source tools. A few of the advantages with Open-Source tools are:

- Free to use and modify

[1] http://www.opees.org/objectives/, April 2012

- Free access to source code
- Possibility to contribute to development

The main advantage with Open-Source is that anyone has the possibility to access the code and continue to develop and maintain the software even if the first author or authors stopped their development.

ArCon has connections to the OPEES project and there were requests from the industry to have support for Open-Source modeling tools in ArCon. ArCon itself is open-source but in its first version ArCon could only check models using the Rational Rhapsody modeling tool by connecting to Rhapsody's Application Programming Interface (API). To truly support Open-Source tools ArCon has to give support to Open-Source modeling tools.

4.1.2 Eclipse and Papyrus

One large Open-Source platform is Eclipse that is a complete IDE (Integrated Development Environment) that enables plugins to extend its usage. Several projects under the Eclipse Group focus on Open-Source tool support. There is a unified effort within the Eclipse Modeling Project[2] (EMP) to provide frameworks, tools and standard for Model-Driven Development. EMP contains several sub-projects that focus on different parts of the goals in the EMP project.

Eclipse and its sub-projects are considered as a mature software development environment that is used in many projects, both commercial and non-commercial. With its increased popularity more people and companies have begun to see Eclipse and plugins for Eclipse as a valid alternative for other commercial tools that have been more traditionally used. As a result Eclipse was set, by Combitech, to be a target platform for ArCon to support, and a specific request to support the Open-Source modeling tool Papyrus, that exist in the Model Development Tools (MDT) Eclipse sub-project, was given.

Consequently a part of this thesis was set to extend ArCon with support for Papyrus, meaning that models created by Papyrus can be checked with the same rules as the first released version of ArCon does with Rational Rhapsody models. Another requirement was set so that ArCon has to be integrated into the Eclipse environment. This means that the model developers can run ArCon from within the Eclipse Editor and check the architecture rules against a model defined in the Papyrus tool. Combitech also requested a preferable solution where the implemented support for Papyrus also enables, or at least makes it easier, to support other modeling tools that follows the UML standard.

[2]http://www.eclipse.org/modeling/, May 2012

4.2 Implementation Overview

To address the requirements, described in Section 4.1, ArCon was extended in two ways. First, an import function was created for the ArCon executable to read UML models defined in a file of the XML Metadata Interchange (XMI) format. Secondly, ArCon was integrated in Eclipse with a plugin. Both ways are described in more detail below under each devoted section, 4.3 respectively 4.4.

The interaction of the ArCon executable and the Eclipse plugin is visualized in Figure 4.1. The Eclipse plugin works as a visual interface for

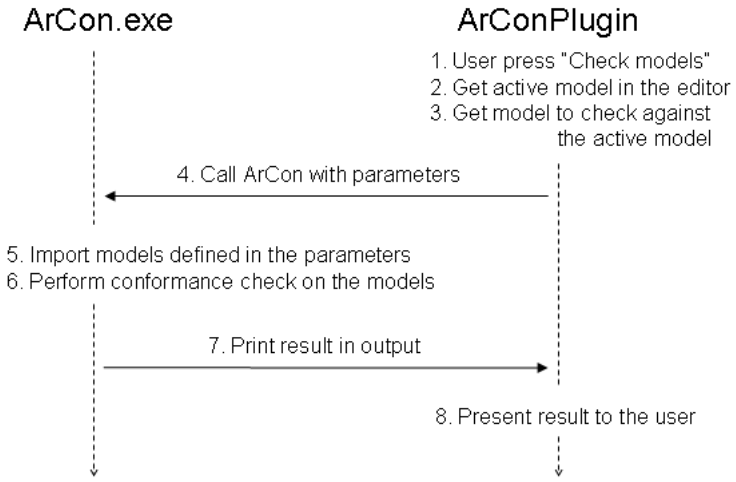


Figure 4.1: Overview of the communication between the ArCon executable and the Eclipse plugin.

ArCon from within Eclipse. The plugin's main purpose is to allow the user to launch the ArCon executable from the Eclipse editor, with the desired models and visualize the results. The ArCon executable has no knowledge about the plugin, which is required to prevent coupling to specific tools. Its only communication is handled through the standard input and output channels, and the plugin serves as the adapter to connect to these channels. The main idea is that other modeling tools should be able to connect to ArCon, with minimal effort, in the same way.

4.3 XMI Import

The XML Metadata Interchange (XMI) is an Object Management Group (OMG) standard for exchanging metadata information via Extensible Markup

Language (XML). It can be used for any metadata whose metamodel can be expressed in Meta-Object Facility (MOF). The most common use of XMI is as an interchange format for UML models, although it can also be used for serialization of models of other languages (metamodels).

The main argument for using XMI as import functionality is that XMI is an international standard, ISO/IEC 19503:2005 Information technology - XML Metadata Interchange (XMI). Using a standard that is independent on the modeling tool avoids the problem where ArCon is too tightly connected to the tool it checks the models for. This was the main problem for the first version of ArCon where it could only read models by connecting to the Rhapsody API. Since XMI is a known standard, other modeling tools can use the extended ArCon as long as they support the XMI standard. With a complete XMI import function, any UML model that is exported to XMI standard format can be read by ArCon. Then, to get fully integrated support, the other modeling tools need only to integrate the call to launch ArCon and read its result. In Papyrus case, this would be the plugin for the Eclipse IDE.

4.3.1 Reading the XMI File

The XMI import functionality requires ArCon to be able to read an XMI file and translate it into its internal meta model classes, which are ArCon's representation of an UML model. The internal classes are currently just a subset of the UML specification with just enough information for ArCon to check the defined architecture rules transformations (see Section 3.3.2 for transformation details). The first step to do this is to read the XML file and translate the XML tags into a data structure that can be queried and iterated. This is done using the Boost C++ library [3], version 1.47, with the *Property Tree* structure. The Boost library was selected for XML parsing for three reasons.

1. Boost is a widely used and recognized library [14].
2. The Boost license encourages both commercial and non-commercial use.
3. Boost library was already used within the ArCon project, and thus no extra dependencies are needed unlike using another XML parser tool.

Boost reads the XMI file and translate it into a tree structure where the nodes contain the XML tags and values according to the schema[4], summarized below.

[3] http://www.boost.org/, April 2012

[4] http://www.boost.org/doc/libs/1_47_0/doc/html/boost_propertytree/parsers.html#boost_propertytree.parsers.xml_parser, April 2012

XML / property tree conversion schema:

- Each XML element corresponds to a property tree node. The child elements correspond to the children of the node.
- The attributes of an XML element are stored in the subkey `<xmlattr>`. There is one child node per attribute in the attribute node. Existence of the `<xmlattr>` node is not guaranteed or necessary when there are no attributes.
- XML comments are stored in nodes named `<xmlcomment>`, unless comment ignoring is enabled via the flags.
- Text content is stored in one of two ways, depending on the flags. The default way concatenates all text nodes and stores them in a single node called `<xmltext>`. This way, the entire content can be conveniently read, but the relative ordering of text and child elements is lost. The other way stores each text content as a separate node, all called `<xmltext>`.

This tree structure is later traversed by ArCon, and while traversing the tree ArCon translates the different objects and values into corresponding meta model classes and connects relationships that together represent the UML model.

4.3.2 Translating Tree Nodes

ArCon must translate the nodes to be able to translate the entire tree structure provided by the Boost library. The design is based on the *Visitor Pattern* [2] where each node in the tree is translated by a specific visitor, which is called a translator. The main reason behind the design decision, to use the visitor pattern, is that the Property Tree structure given by the Boost library is very suitable for the pattern. The goal is to translate each UML element into a class in ArCon, that corresponds to the given element, thus it is convenient to assign one visitor for each element type that needs to be translated. The visitors follow the *"do one thing and do it well"* methodology by letting each visitor only be responsible for translating one specific UML element type. The approach is also appropriate for future support where the need to translate new UML elements and properties can emerge. New visitors, that are responsible for new UML element translations, can then be created and implemented in the code without any or small changes of the current code base.

Two obstacles were identified when designing the code structure according to the pattern described above.

- First, UML elements may inherit properties from other elements according to the specification but, according to common practices, different parts of the code should not be duplicated when doing the same

operation. In this case, different visitors should not have duplicated code for translating the same type of nodes, for example the name of the element.

- Secondly, the correct visitor has to be used on the correct nodes. The correct nodes mean more specifically the nodes where the UML element, that the visitor is responsible for translating, are described in the tree structure.

The first problem can easily be solved in an object oriented programming language like C++ where you have inheritance just like in UML. The visitors can inherit another visitor and ask it to translate nodes they can not translate themselves. The approach is the *Chain of Command* [2] design pattern, and the call chain can go all the way down to visitor manager which is the base of all translators.

The second problem was solved by introducing the visitor manager, which was earlier referred to as the base of all translators. This manager has knowledge of what is described as BaseNode in this report. BaseNode is a node that contains an element and its packedElements (child elements) [5] within its sub-tree. The visitor manager can recognize these nodes and delegate the work to the visitor which is responsible for translating the UML element represented in the node.

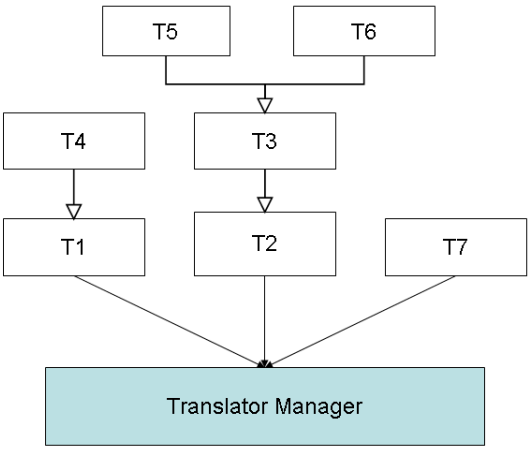


Figure 4.2: An overview of the translator layout.

Given a current translator that does not know how to translate a node, which in this case is a BaseNode, then the call stack will go all the way down to the visitor manager. In Figure 4.2 the call chains can be visualized by following the inheritance and association down to the Visitor Manager.

[5] see UML specification for the specification of packedElements

The visitor manager will then recognize the BaseNode and change the active translator and delegate the translation of the BaseNode to another translator which can translate it. If no such translator can be found a 'No Translation' error is reported instead.

4.3.3 Translator Design

The design principles described in 4.3.2 resulted in two main visitor classes.

- **BaseTranslator**
  BaseTranslator is the main translator, also called visitor manager or translator manager, and is responsible for translating the entire tree. There exists only one BaseTranslator and all SubTranslators are connected to the BaseTranslator. BaseTranslator is a translator manager which only delegates work to its SubTranslators by recognizing BaseNodes and letting the responsible SubTranslator do the actual translation.

- **SubTranslator**
  Each SubTranslator is responsible for the translation of one specific class of ArCon's meta model classes. The SubTranslators can inherit from another SubTranslator in the same way as the meta model classes inherit from each other. The inheritance enables the SubTranslators to ask their baseClass for translation if the node contains values, defined by the baseClass of the meta model class, that the SubTranslator is responsible for translating. This call stack can go all the way back to the BaseTranslator if no baseClass can translate the given node. The BaseTranslator can then see if the node is another element owned by above element, typically packedElement, and then delegate the translation to another SubTranslator, which is responsible for translating that particularly element.

The facade pattern [2] is used on the construction of the translators by defining of few interfaces which the translators must implement. The interfaces are constructed in a hierarchy accordingly to Figure 4.3.

- **ITranslator**
  Interface shared by all translators. Defines some basic function calls that all translators need to supply.

- **IBaseTranslator**
  Interface for BaseTranslator which contains function calls that only the visitor manager should implement.

- **ISubTranslator**
  Interface for SubTranslators which contains some code for connecting to a BaseTranslator in a proper way. All new visitors should have ISubTranslator as one of its inherited classes.

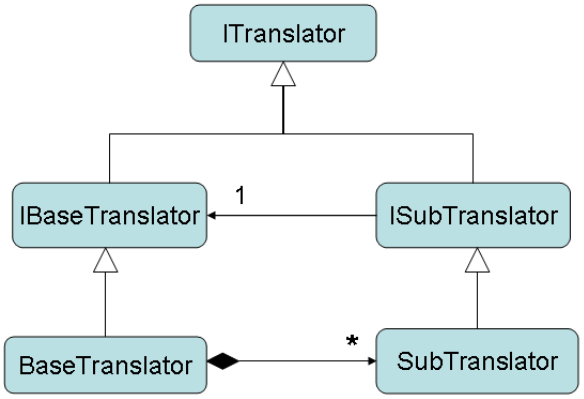


Figure 4.3: An overview of the translator interface layout, and the classes implementing them. Classes denoted with "I" at the beginning are interfaces.

- **SubTranslator template**
  Template class that has some comfort methods to create new SubTranslators.

4.3.4 Main Translation Loop

The main loop, when translating the tree containing the UML model, is TraverseTree which is defined in BaseTranslator and a snippet of it looks like following:

```
1. TraverseTree(Element* owner, Tree tree, ITranslator* translator)
2. {
3.   for each node in tree
4.   {
5.     NodeResult nodeRes = translator->TranslateNode(owner, node);
6.
7.     ITranslator* newTranslator = nodeRes.getTranslator();
8.     Element* newElement = nodeRes.getElement();
9.
10.    //Check result
11.    if( nodeRes.HasNewTranslator() )
12.    {
13.       //translate rest of the node with another translator
14.       TraverseTree(newElement, node.second, newTranslator);
15.       newTranslator->PostNode(newElement);
16.    }
```

```
17.     ...
18.  }
19. }
```

As can be observed in the code snippet above the function *TranslateNode*, on line 5, returns a *NodeResult* object. The purpose of this object is to tell the BaseTranslator how it should continue to traverse on a node, and which translator to use in the continued translation. Four different types of answers can be given by the NodeResult:

1. **Continue**
   Continue into the tree from the node using the same translator that translated the affected node.

2. **Continue with new Translator**
   Continue, but use the new translator given in the NodeResult object.

3. **Node is finished, stop traversion**
   The node is finished so stop traversing further into this node.

4. **No translation exist**
   The affected node could not be translated by the currently assigned translator or its baseClasses. Neither did BaseTranslator know who to delegate the work to. This happens either if the model contains elements that ArCon do not support or if a node contains XML info that is not of ArCon's interest.

4.3.5 Connecting References

Most of the translation is done with this in-place approach meaning that corresponding objects are created on the fly when traversing the tree, using the *TraverseTree* function. Values, properties and other values are added to the objects while traversing the attributes of objects in the UML tree. This works for the most part, but everything can not be connected or created on the fly since some references may refer to elements which have not yet been traversed and created at the time visiting the reference. To address and solve this problem two other important functions are part of the translation interface. These functions are:

- **PostNode**
  Is called when a BaseNode is finished and it allows the affected translator to know when all subnodes have been visited. This enables relations and values to be connected to objects that are created within subnodes further down in the branch.

- **PostTranslation**
  Same design principle as PostNode but PostTranslation is called after the entire tree has been traversed. The main purpose is to connect

relationships and other connections that refer to different elements, that may not yet have been created when the relationship is found during the tree traversion. Two different elements may be created in any order, depending on where they are defined in the XMI file, and as a result the connections between the two objects can only be done when both have been created. By saving the reference ID and wait until after all objects have been created, it can be assured that all elements, to be created, will have already been created. This means that the connections can be done without any risk of referring to an object which has not yet to be created. Each SubTranslator is called in order of their hierarchy order to ensure that all dependencies are set in the correct order.

The PostNode function is called within the *TraverseTree* loop, and is visible in the code snippet on line 15 at page 25, while PostTranslation is called after the function *TraverseTree* has finished.

4.4 Eclipse Plugin

The purpose of an Eclipse plugin for ArCon is to integrate ArCon into the working environment for the model developers. The plugin will serve as a front end for the user, so they will not have to leave the model editor to check the model. By just pushing a button the developers or architects will be able to launch ArCon to check if the architecture model conforms with the system model. A few core obstacles to be solved were identified during development:

- Find out which two models the user actually wants to check conformance on when launching ArCon using the plugin. The main problem being that only one model is visible in the editor so how can the other, related, model be found without having to ask the user every time.
- The plugin only works as a front end for the ArCon executable and thereby it should be easy to update the plugin with new updates for ArCon. The plugin and ArCon executable do not share the same development platform and as a result ArCon itself can not be compiled into the plugin.
- The executable could be packaged into the plugin, but it was recognized as unnecessary to also have to update the plugin each time the ArCon executable would be updated. Consequently the plugin and executable had to be separated. The plugin would still have to call the executable and receive its result.
- When introducing the XMI import functionality ArCon got the requirement to support two different ways to read UML models. One

for reading from Rhapsody projects and one for reading from an XMI file. At execution ArCon has to know which way to read the models. This was addressed by letting the input parameters to the executable tell ArCon which reading method that is supposed to be used. The plugin has to supply these parameters when calling the executable.

The obstacles and functionality requirements described were addressed according to the description provided in the following subsections within this section, 4.4.

4.4.1 Finding Models to Check

The active file in the editor is considered to be one of the target models (system or architecture). A three step system has been developed, visualized in Figure 4.4, to find the related model (architecture model if the active file is the system model, and vice versa). First the plugin checks if there is any saved file containing info about the related model for the active file. The saved file structure holds information about the path to the related model and what kind of model it is (system or architecture model). Second, if no such file exists yet, then the plugin will try to find the related model using a naming convention. The naming convention is defined as both models have the same name but they have a different affix, currently prefix or postfix. The default is to have a postfix "_arch" on the architecture model and a postfix "_system" on the system model. Using this convention the plugin can find both models from just knowing one of the models when the user launches the operation. The last step to find the related model, if none of the above worked, is to ask to user to select the related model via a file browser. The selected model will be saved in an info file, which will be used next time to find the related model accordingly to step one.

4.4.2 Finding ArCon Executable Path

The user will be asked to supply the path to the ArCon executable on the first time running the plugin. This is done using a standard file browser. Given a valid executable the path will be saved in the plugin settings. The same executable will be used for all later executions unless the path is manually changed through the plugin's preferences page in Eclipse.

4.4.3 Run ArCon with Parameters

Using the supplied path the ArCon executable is launched with a set of parameters. The parameters are given in the following order and form:

1. [**mode**] Mode must be the string "UML" when using XMI import to read the models. If Rational Rhapsody is supposed to be used then the string must be "Rhapsody", and the rest of the parameters are ignored.

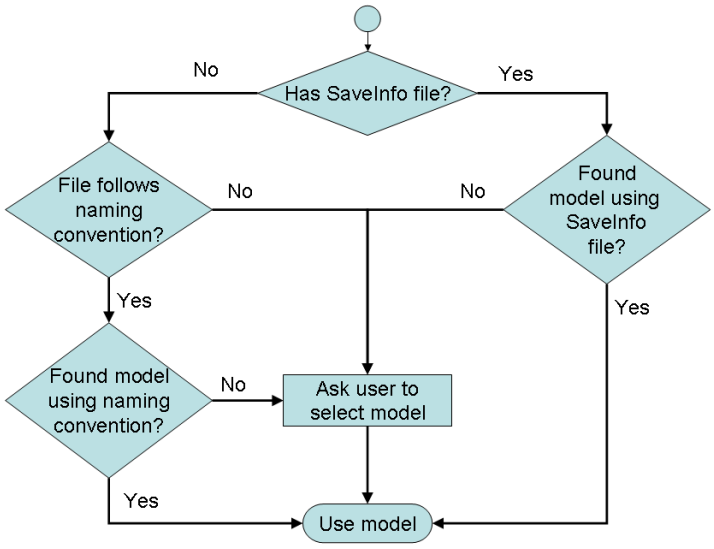


Figure 4.4: Flowchart over the retrieving of the related model to an active model.

2. [**architecture model path**] String containing the absolute path to the architecture rules UML model file.
3. [**system model path**] String containing the absolute path to the system UML model file.
4. **"debug"** (optional) If the fourth parameter is set to "debug" then the fifth parameter will be read to set the debug level.
5. [**debug level**] (optional) If "debug" is set then this parameter defines the debug level when reading the XMI file.

The first argument, "UML", tells ArCon to use XMI import to get the architecture and system model using the paths defined in the next two arguments. There are also two optional parameters at the end which allows ArCon to be run with extra debug settings when reading the XMI file. The settings are used to debug if any error or problems occurred during the reading of the XMI file, and to give information on what the problems were caused by. Debug level is a numeric value according to:

1. **No Debug**
   No extra debug messages will be printed by ArCon.
2. **Only Errors**
   Only serious errors that hinder the programs continued reading and translation of the UML model are reported.

3. **Errors and Faults**
   Errors and faults that are known to be wrong are reported. Faults are not as serious as errors.

4. **Error, Faults and May-be-wrong**
   Errors, faults and occurrence in the code that may be wrong are reported. Reports of type May-be-wrong are mainly things the program could not translate. This can be things that are not of interest of the program, for example extra XML tags. But, in case of a bug, it can also be things that the program should translate. From the program's point of view it can not differentiate from the two different cases, hence the may-be-wrong.

5. **Error, Faults May-be-wrong and Notifications**
   Using this setting let ArCon give detailed report about errors, faults, may-be-wrong and its progress when reading and translating the UML models.

The ArCon plugin will supply all parameters when launching the ArCon executable. The model paths are retrieved using the method described in Section 4.4.1. Debug settings are set to default 1=Only Errors, but it can be changed manually from the plugin's preference page within Eclipse. The settings in the preferences page are visualized in Figure 5.2 on page 33.

4.4.4 Receive and Present Result

The result from ArCon executable is returned on its standard output. The Eclipse plugin uses this by connecting to ArCon's output and retrieving the result from there. Given the result from ArCon, the plugin will present the result for the user by printing it into the console in Eclipse.

Chapter 5

User Guide

This chapter will show new users how to get started with the ArCon plugin and how to check conformance, using ArCon, on models created by Papyrus in the Eclipse editor.

5.1 Getting Started

5.1.1 Installation

The tools needed are Eclipse IDE, Papyrus, ArCon executable and ArConPlugin.

Install Eclipse Download and install Eclipse IDE from the Eclipse homepage[1]. Start Eclipse using the eclipse executable.

Install Papyrus In Eclipse, go to Help/"Install New Software..." and select the Indigo update site. If the Indigo update site[2] is missing then add it using *Add repository*. Install MDT Papyrus under the Modeling category on the Indigo update site.

Download ArCon.exe Go to ArCon project page[3] and download the ArCon 1.5 executable. This is the executable to be used by ArConPlugin later on.

Install ArConPlugin On ArCon project page also download the ArConPlugin v1 for Eclipse. Extract the archive into the eclipse installation folder. The .jar file should be located in the *eclipse/plugins/* folder.

[1] http://www.eclipse.org/downloads/, May 2012
[2] http://download.eclipse.org/releases/indigo, May 2012
[3] http://code.google.com/a/eclipselabs.org/p/arcon/downloads/list, May 2012

When all steps are completed then restart Eclipse. The plugin should now be running and UML models can be created using the Papyrus modeling tool.

5.1.2 Plugin Interface

The ArCon plugin comes with a menu, visualized in Figure 5.1, where the plugin settings can be changed and an ArCon executable can be launched to check the conformance on an active model in the editor.

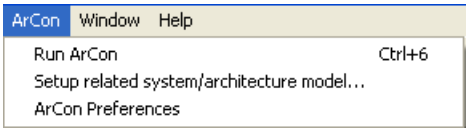


Figure 5.1: Screenshot of ArCon plugin's menu.

- **Run ArCon**
  Calls the ArCon executable and checks the conformance on the active model and its related architecture rules or system model.

  The active model is considered to be the UML model currently visible in the editor. If the plugin is launched for the first time then the user will be asked for the path to the ArCon executable. Supply the path to ArCon 1.5, or newer compliant version, if asked to do so. The user may also be asked to supply the related model if the related model is not obvious from the plugin's point of view, see Section 4.4.1 for details on how ArConPlugin finds the related model.

- **Setup related system/architecture model...**
  Menu item to explicitly define or change which model is related to the *currently active model*. Using this option will override the soft detection, like naming convention, and always use the specified model as related model.

  An important note is that the related model may not be the same for both models, which means both the architecture and system model may not have each other as related model. This behaviour is intentional since there is no guarantee that an active system model may always be refereed to from the architecture model. The reason is there may be several system models that use the same architecture rules, and the architecture model can only have one related model. All the system models can still have the same architecture model as their related model.

- **ArCon Preferences**
  Menu item to change settings for the plugin. On the preference page

the user can change the path to the ArCon executable which is most often changed if the ArCon executable is moved in the file system or another version is supposed to be used instead. Other settings like *Debug level* or naming convention can also be changed on the preference page. More detailed description about the usage of the Debug level settings can be found in Section 4.4.3. The settings presented on the preference page can be seen in Figure 5.2.

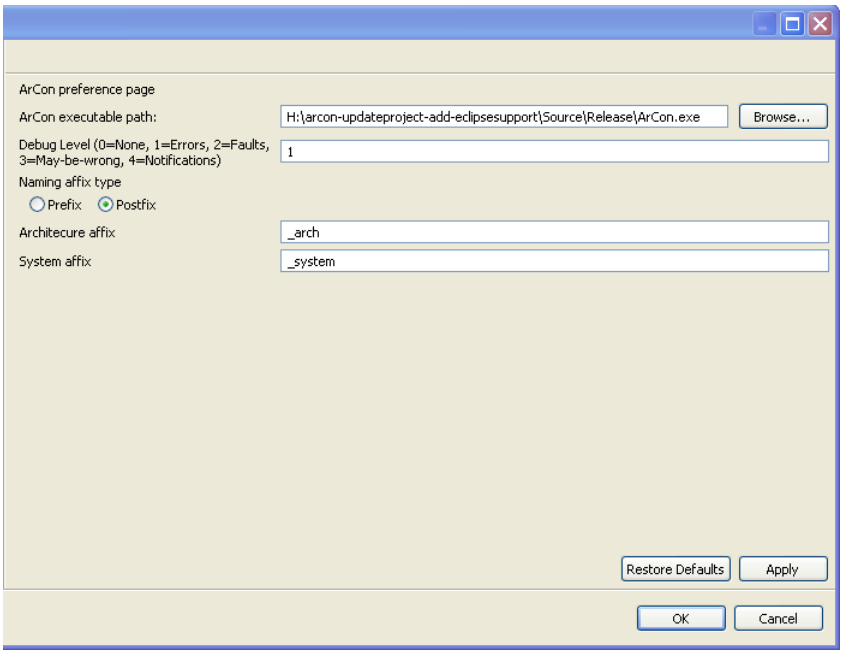


Figure 5.2: Screenshot of ArCon plugin's preferences page.

5.2 Model Examples

This section will give some basic examples on how to design the architectural rules model and the system model. It is important to note that classes in the Architecture Rules Model translate to stereotypes in the System Model, for more detailed information on how the translation works see Section 3.3.2. Therefore an UML profile, which contains all the stereotypes that are defined and constrained in the architectural rules model, has to be created and the system model should apply the profile and use the defined stereotypes.

5.2.1 Example1

The example, shown in Figure 5.3, shows how the architectural rules can be used to constrain which types of classes a package may contain. The

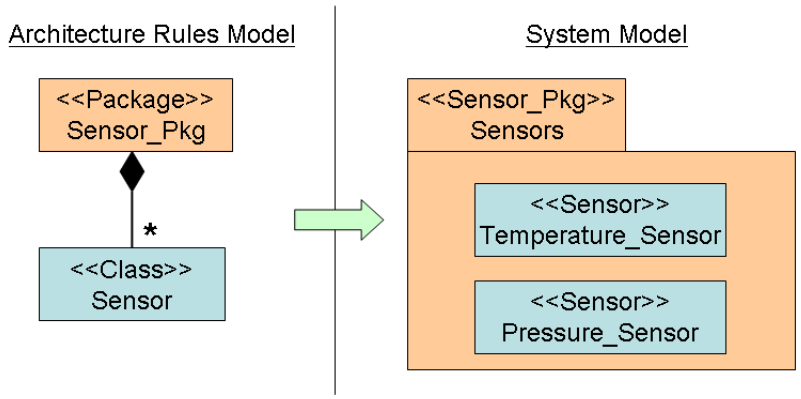


Figure 5.3: Example of how architectural rules translate to the system model. The colors visualize the transformations.

architectural rules model contains two classes, Sensor and Sensor_Pkg, which have the stereotypes `<<Class>>` and `<<Package>>`. Between the classes is an association of the type composition drawn. The composition multiplicity has the value [0..infinity], defined by the '*'.

The rules and constrains defined by the architectural rules model in the example are the following:

1. A package with stereotype `<<Sensor_Pkg>>` may contain any number of classes with stereotype `<<Sensor>>`.

2. No other elements are allowed in a `<<Sensor_Pkg>>` package.

The first rule is defined according to the T8 tranformation rule, see Table 1. The second rule is applied because of the general rule by ArCon that nothing that is not *explicitly* allowed is allowed.

The stereotypes on the classes, in the architectural rules model, define the element types that the translated stereotypes are to be applied to, in the system model. In this example it is realized by the Sensor class with stereotype `<<Class>>` being transformed to a stereotype named `<<Sensor>>` which can be applied to classes, and the Sensor_Pkg class with stereotype `<<Package>>` being transformed to a stereotype named `<<Sensor_Pkg>>` which can be applied to packages.

On the system model it can be observed that these constructs exists in the form of a package named *Sensors* which has the stereotype `<<Sensor_Pkg>>`

and two classes, Temperature_Sensor and Pressure_Sensor, with the stereotype `<<Sensor>>`. Because of the multiplicity value [0..infinity] there are allowed to exist more than one type of `<<Sensor>>` classes within the package, and as such the two different sensor classes are allowed.

5.2.2 Example2

In Example2, shown in Figure 5.4, it can be observed how the system model can violate the defined architectural rules.

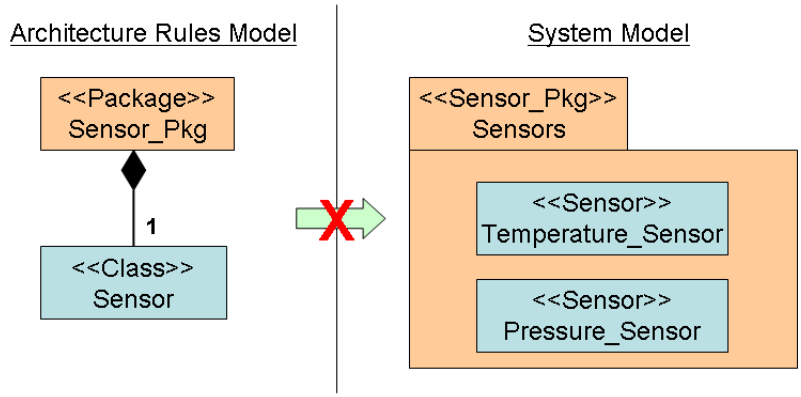


Figure 5.4: An example how architectural rules can be violated in the system model.

Example2 is exactly like Example1 except that the multiplicity on the composition, in the architectural rules model, has the multiplicity 1. This means that only one type of `<<Sensor>>` class is allowed in a `<<Sensor_Pkg>>` package in the system model. In the example, the Sensors package in the System Model contains two different types of classes with stereotype `<<Sensor>>`, and it is not in align with the defined architectural rules. Therefore an error, marked with a red box in Figure 5.5, would be reported if a user executes ArCon to check the conformance on the two models.

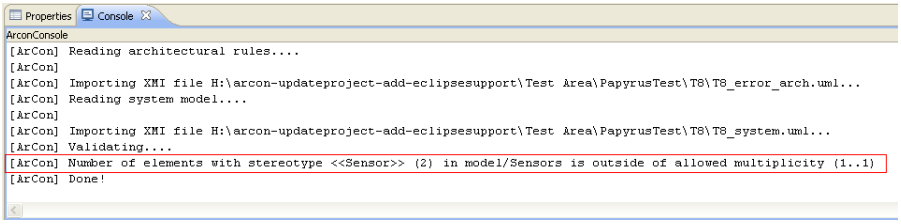


Figure 5.5: Example how a validation report given by ArCon looks like. An error has been marked with a red box.

Chapter 6

Method Evaluation

This chapter will go into how the method to model architecture rules, described in Section 3.3, works in practice by looking into two case studies. First the result from an earlier study, performed by the authors of ArCon, is summarized. Later the case study performed within this thesis project is presented and last some reflections are reported.

6.1 Earlier Study

The authors of ArCon performed a case study on their suggested method to model architectural rules [11]. Their selected system was a software platform for digital TV set-top boxes for the DVB[1] standard. One objective with their study was to demonstrate the applicability of the approach, by modeling the architectural rules according to the approach in, an existing, previously developed system. The main goal of this activity was to establish the degree to which it enabled modeling of the architectural design rules. They selected the project on the following criteria:

- The system had to have been developed using MDD.
- The system had to be an existing real system of significant size and with a sufficient functionality to make it generally representative as a real-world embedded system.
- The architecture, including the architectural design rules, had to be documented to a level where it could be interpreted by the research team.
- The research team had to have good access to people who had first-hand knowledge of the architecture, to be able to see beyond the documentation and to be able to resolve any ambiguities.

[1] Digital Video Broadcasting, http://www.dvb.org. April 2012

The project was implemented using the modeling tool Rhapsody (version 4.x) from Telelogic [Telelogic Rhapsody modeling], with all code generated from UML models in the tool, using C++ as the action code language. The size of their software platform was approximately 350,000 eLOC in C++ and the effort to develop it was about 100 person years over a 24 month period. The architecture was documented partly in the system model and partly in one manually written document. The system model contained a high-level package structure and a framework of classes supporting the architectural design rules. The document contained the architectural design rules. The researchers had first-hand knowledge of the architecture, since the primary author of the article was the technical manager of the project, responsible for work practices and tools. The architecture was, however, developed by two other persons acting as architects. The study was, according to the authors, conducted by a systematic walkthrough reviewing the rules from the architectural document in several iterations, gradually transforming them to modeling constructs according to the proposed approach.

The authors of the article stated some questions to be evaluated in the study.

1. To what extent could the specified rules be modeled?
2. Were there certain kinds of rules that could not be modeled and if not, why not?

These questions are very interesting since they give a very basic answer to how applicable the approach is to model architectural rules. In their study they had 66 rules to be defined and of these 66 rules only 8 could not be modeled. All these eight rules were considered to be of the type where the developers were supposed to exercise judgement, which made them inherently impossible to formalize.

At the end of the article the authors stated that although the approach has only been tested on one system, two factors suggest that the results should, to a large extent, be transferable to other systems and organizations in the embedded software domain.

1. The defined transformations are based on raising the general modeling constructs of UML to the meta-model level, not on the specific needs of the system used for the test.
2. It is a real-world embedded system of significant size with functionality quite common in this domain.

The authors also state that more practical studies are needed to better evaluate the implications when adopting the method in different domains.

6.2 Case Study

This section will present the result from the case study performed in this thesis at Saab Aeronautics[2].

The selected project was an active industrial project within the Saab JAS Gripen product, which is a fighter air plane. The evaluated system model was a SysML model created in Rational Rhapsody 7.6.1. The project is of type software engineering with a lot of diagrams and elements on the behavioural constructs of UML/SysML. This made the project interesting since ArCon's earlier studies have been performed on architecture rules existing within the static side of UML with Structural constructs, and more specifically the class diagrams. A document containing guidelines and modeling rules for the project was available and served as a base for the architecture rules. Using the document together with a few project representatives a few core architecture rules and problems were identified, by the Saab stakeholders, to be evaluated whether they can be automatically checked by ArCon.

During a workshop the identified rules were discussed to find out which were feasible targets for ArCon. The rules were graded with three types:

- **Working** Rules graded by this type could be checked in the current implementation of ArCon, version 1.5.
- **Possible** These rules can be checked in a near future if ArCon receives a small update.
- **Not-Feasible** A major update is required for these types of rules, or the rules are not of a type possible to be checked by ArCon.

Saab's list of rules, that they wanted to be checked, contained 26 rules of different types. Of those 26 rules, two rules were identified as *Working* and nine were identified as *Possible*. The two *Working* rules were one rule for having a specific prefix on a stereotype and another rule for allowed associations between different stereotypes. The *Possible* rules could most often not be checked, with current ArCon, because of a lacking representation of the elements in ArCon. For example, rules on dependencies between Use-Cases could not be checked because the representation of Use-Case did not exist in ArCon. The *Not-Feasible* rules contained a wide range of different types of rules, but a few core problems could be identified:

- **Diagrams**
  A big category containing non-feasible rules were diagrams and a wide range of rules involving diagrams were identified. A few examples of rules were diagram naming conventions, implicit rules on elements visible in a diagram and rules on the diagrams that must exist for an

[2]http://www.saabgroup.com/, May 2012

element type. For example two elements had an implicit relationship, defined by the project holders, if they existed in the same diagram. The problem for ArCon is that it has no knowledge of the diagrams but only of the UML elements and their constructs, such as attributes and operations.

- **Conditional Rules**
  Many rules were conditional, meaning a certain structural pattern or other type of pattern had to occur for the rule to be applicable. This rule matching was contrary to how ArCon matched rules by identifying stereotypes and applying the rules on the elements using the stereotypes. An example where stereotypes were used, but still had a conditional rule, was that a block with a specific stereotype could only have associations navigable either to it or from it, not both directions. This could be fixed by specifically splitting the blocks into two stereotypes, `<<To>>` or `<<From>>`, but the stakeholders wanted to avoid changes of the current stereotypes as much as possible.

  Another example was that a Block, of a specific stereotype, can only have associations to other Blocks of the same type and on the same hierarchy level. The example is visualized in Figure 6.1, and as can be observed the associations can only be called on the same hierarchy level or to a top level. Using stereotypes, in this context, offers little flexibility, and if the rules would have been implemented with the available version of ArCon, version 1.5, all levels in the hierarchy would have to be stereotyped in the architectural rules model. Not only would it lead to unnecessarily many stereotypes but it also requires the architect to explicitly know the class hierarchy on the system model. A knowledge some may say the architect should not, or can not, have beforehand. The architectural model would also have to be updated if the hierarchy would be extended during development or in a future update.

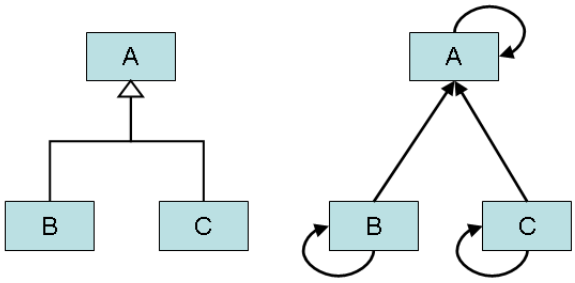


Figure 6.1: Conditional Rule. Left diagram shows the class hierarchy and right shows the allowed association paths.

- **Several Steps or Depth** Some rules between two stereotypes could not be checked because their relation was connected through a chain of elements. Sometimes the relational chain could consist of several steps of explicitly or implicitly defined element relationships. Implicitly defined rules were typically a said relation if the element existed in a certain diagram. An example of a short relational chain is a *Block* of stereotype `<<MyBlock>>` that must have *Ports* with a connected *FlowSpecification* with stereotype `<<MyFlowSpec>>`. In this example the Block and FlowSpecification have a rule together, but they are not directly in connection with each other since the Port acts as the connection between them. Since the Port does not need to have a specific stereotype it creates a chain that could not be checked by ArCon, since it can only identify stereotypes and apply rules connected to them.

  Another example was that a rule on elements contained in a package, of a specific stereotype, had to be applied on all levels within the package. The root package had the stereotype which defined the rule and all elements within the package and its sub-packages should be constrained by the rule. The problem was that ArCon could not handle a generic depth of the package structure and as a result could only force the rule on the elements directly under the root package.

- **Generic Rules**
  A few rules were of generic type, where the rules would apply to element types and not on stereotypes. For example there were some general rules on all Use-Cases and their dependencies. The transformation rules defined for ArCon, see Table 1, contain a basic definition on how to define element meta types, as well as stereotypes. But the element meta type definition has no support in current ArCon, marked as 'Not Supported' in Table 1, and can thereby not be used yet.

- **Naming Convention**
  ArCon contains a basic naming convention mechanism for stereotypes but it lacks a proper regular expression method. Some of the naming rules, requested in the case-study, were constrains on string length, upper- or lower-case on different parts of the string, word separation with underscore, affixes from a list of allowed words, enumerations and name relations to diagrams.

Due to administrative complications the case-study could not be finished within the timeframe of this thesis. Regardless, the workshop gave insight on ArCon's potential and its current issues. These insights are presented in the next section, Section 6.3.

6.3 Analysis

In this section the insights and reflections received during the case-study, and from reading the results from the first case-study performed by the ArCon authors, are presented. If no explicit definition is present, *the case-study* is referring to the case-study performed within this thesis that is presented in Section 6.2.

The two different case-studies show how different the results can become when performed on two widely different software projects. The first case-study, performed by the authors of ArCon, was performed on a software projects where rules on the static structure and outline, such as classes and their relationships, of the system was a main goal of the architecture conformance. On the other hand the second case-study was on a project that had no earlier knowledge of ArCon, and its methods to define and constrain architectural design rules. Therefore the rules to be checked, which were requested by the project owners, had no pre-condition to what ArCon was created for. As a result the case-study, performed in this thesis, was interesting because it showed how ArCon performs in a project for which it has not been tailored for, contrary to the first case-study.

Practical to Use The presented numbers in the first case-study, performed by the ArCon authors, gave good answers for the questions stated in the article. A large number of rules could be modeled and checked with ArCon, and the rules that could not be modeled were of a type that required judgement. Although, one could question the lack of analysis how practical the approach is in the sense of understanding and usability for the architects and developers. The questions above only answer whether the approach can be used to define and check the rules but state nothing about how useful and easy to understand the approach really is. In the report their main reason for developing this approach is they found no intuitive and easy-to-use approach to model architectural rules, which can be automatically checked against the system implementation which they call system model. One of the current practices, which they evaluate in their report, is using OCL (Object Constraint Language) to constrain the system model. A drawback, with using OCL, stated in their report is how complex the rules definition can become. This complexity makes it impractical and require beyond what can be expected knowledge from a typical architect or developer. After stating this disadvantage the report tries to put the approach in light as an easy-to-use solution without really answering how practical it is. Short examples are shown in the article which presents how their approach would compare to the OCL examples, which in their state give the viewer the impression of a more intuitive method. The report leaves much room for questions on how practical the method is for outsiders, who have not been presented with the approach before. This is especially of interest since the persons involved

in this case study are also the creators of the approach, or have at least had contact and guidance on the method from the creators beforehand. Especially one question about the notion of 'intuitive and easy-to-use' arises when reading the description of the approach:

- Architectural rules model is modeled using standard UML syntax, but the meaning is at the higher meta-model level.

While using the standard UML syntax has the advantage of allowing the architectural rules model to be modeled using existing modeling tools without extension, it also risks to be confusing for the user. It may be confusing to read and understand what the rules really mean when the architectural rules model uses the same syntax as the standard UML models while having another meaning. Therefore other interesting questions arise:

- How easy to understand is the architectural rules syntax and meaning? Will it require training for a typical architect or developer?
- Will using standard UML syntax, while having another meaning than the standard, cause confusion for the architects and developers? If so, will confusion only happen when the subject is new to the method, or will it be confusing in other situations, for example when switching between working on standard UML models and architectural rules model?

For the method to be widely used and accepted, one could state that it must be easy to pick up and use without any experts' guidance. Consequently more case studies would have to be performed to evaluate the methods applicability within software development. The first case study gave a hint on its potential and gives the reader a preview of how the method can aid the software development. But, as also stated by researchers, there is a need for further research to study the implications when adopting the approach in other application domains. Factors to investigate include the ease with which architects, developers and other stakeholders can learn the approach and accommodate their working practices to it.

Different Levels of Architectural Rules One interesting insight obtained during the case-study, preformed within this thesis, was the need for other levels of architectural rules. Early in the study it was obvious that different levels and definitions of architectural rules exist. Several rules existed in a category that could be called *Modeling Rules* or *Modeling Practices*. These types of rules were more general rules or guidelines on how to model systems, and could be transferred to other projects since they were not specific for the project. The rules could often be seen as the modeling equivalent of programming standards and practices. Rules on how to use diagrams and how elements are connected to diagrams were common. ArCon currently focuses on project specific rules that define the realizing system

model, and its structural design and constrains. As a result the scope of ArCon does not fall under the *Modeling Practices* rules, even if the rules are still identified as of interest. A proposed solution is to create a sister-project for ArCon that checks the non-project specific types of rules, rules that were earlier called Modeling Practices rules.

Element Representation One big drawback with ArCon's current implementation is the lack of representation of many UML elements. ArCon has a representation for a small subset of UML's entire set of elements, but to check rules on an element type it needs to have a representation in ArCon. Therefore more element types need to be added in ArCon, before the projects using those element types can be properly supported. Examples of elements identified as of high priority, and set to be implemented during the case-study, were *Use-Case*, *Port* and *Flowspecification*.

Mapping Technique Another drawback is the mapping technique, between the constructs in the system model and the architectural rules model, that is currently only composed of stereotypes. The technique works well for certain types of problem, like defining a design pattern, but it can also lead to a problem, in this thesis, called *stereotype explosion*. Stereotype explosion means that a large number of stereotypes need to be added on the elements in the system model, to be able to check the rules. A risk is that the model and elements receive too many stereotypes such that they become messy or unintuitive. The stereotypes added on the elements should be intuitive and not forced upon just so that a rule can be checked. Consequently, and as seen in the case-study, there is a need to add more ways to map the rules, defined in the architecture model, on the system model. Some of the needed techniques are using general element types and defining smaller relational chains for the rules, where the chain can consist of general element types without a specific stereotype.

Conditional Rules Different types of conditional rules, that could not be described using stereotypes, were also needed to define some rules during the case study. There were also cases were stereotypes could be used, but it would require unnecessary many stereotypes to define the rules. It could also lead to that some stereotypes had no real meaning, or use, except being able to check the architectural rule. Consequently, a way to define conditional rules was requested as a possible solution to solve the described issues.

Stereotype Instances In the architectural rules model there may also be a need to refer to an element instance of an applied stereotype. For example a class *TempSensor* that applies the stereotype `<<Sensor>>` would be an element instance of the stereotype `<<Sensor>>`. The architect may, as an example, want to say that all classes with stereotype `<<Sensor>>` must contain a function that has their own type as return type. This would mean

a TempSensor would return a TempSensor object in its function, and other `<<Sensor>>` classes would return their type. With the current transformation definition, see Table 1, there is no possibility to define such a rule. The closest one can get is to define that the function must return an element type that has the stereotype `<<Sensor>>`. In this example it would mean TempSensor can return any type of element that has the `<<Sensor>>` stereotype. As a result it would open up the system and not constrain the system enough, according to the architect's needs.

A possible suggestion could be to introduce a new stereotype, `<<Instance>>`, and transformation rule in the architectural rules model. Applying the stereotype `<<Instance>>` on a class in the architectural rules means the class is an instance of the class it inherit. Using the instance class as a type, for example return type for an operation, within the architectural model means an instance of the stereotype is referred, and not the stereotype itself like the normal syntax. The problem example, with TempSensor, could be defined in the architectural model accordingly to Figure 6.2. The `<<Sensor>>` class defines that each class that applied the `<<Sensor>>` stereotype must have an operation named *oper1*, with no arguments and returning its own class type.

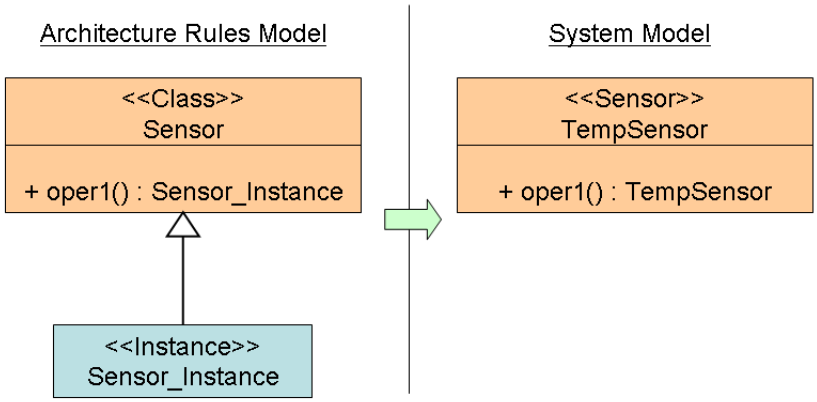


Figure 6.2: Possible syntax for Class Instance reference.

Summary The workshop showed the potential for ArCon while it also gave insight on a number of issues and where there is room for improvement, most likely because the project was so different from ArCon's first target project. Even though many of the rules could not be checked right from the start, there is potential to check them in the future when ArCon and its method have evolved. Since the project, of the case-study, was in a domain which was not part of ArCon's first target, it can be expected that new features and fixes are needed when expanding the system's domains. Some

of the issues presented in this section are feasible to implement directly, while others do not have a straight forward solution and require more thoughts before they become applicable. A few of these improvements were set to be implemented during the case-study, but it could not be performed within this thesis because of time constraints. Instead, the ideas and suggestions are presented in Section 7.2.

Chapter 7

Summary and Conclusions

This chapter presents the results for this thesis project and gives suggestions on future work and improvements for ArCon.

7.1 Results

The software architecture, with its rules and constrains on the system model, is important in many projects. To check and maintain the conformance of the architectural design rules, on the realizing system, is a problem within software development. ArCon uses a proposed method to define and check architectural design rules defined in a UML model. In this thesis, ArCon was successfully extended with import functionality so that ArCon could read and check UML models defined in an XMI file. A plugin was also successfully created to integrate ArCon into the Eclipse environment, and Papyrus as the first modeling tool in the Eclipse platform to be supported.

The method itself, which ArCon builds upon, was evaluated in a small case-study. Even though the study was performed in a very short time frame it gave light to some of ArCon's potential and shortcomings. Several potential areas to improve ArCon, and the method itself, have been identified during development and will be presented in the next section, Section 7.2.

7.2 Future Work

This section will give some suggestions of the possible extensions and further development of both ArCon and the proposed method, to define architectural design rules, used by ArCon.

7.2.1 Method Evaluation

The proposed method is very new and untested in the current state. More practical case-studies from different areas within the industry, that use

Model-Driven development, need to be performed and evaluated to give a true grade of the method's usage. A lot of different aspects need to be considered when evaluating the method, and not only whether the method can define a specific rule or not. To be able to define a rule is not enough since it has to be intuitive both when defining and reading the rules. The lack of intuitive and easy-to-read methods was one of the core reasons for the method to be developed in the first place. Using OCL constraints was one of those methods that seemed unintuitive and hard to understand that is used in projects today.

Other aspects also need to be considered, such as how the method fits into a company's or project's practices and processes. Some examples of interesting questions for a company or project are:

- What are the benefits with the method?
- What are the drawbacks with the method?
- How does the method fit into the current processes?
- Which rules, or what types of rules, can the method check that currently exist, in different documents and artefacts, within the project?
- What rules that ArCon can not check would be feasible to do so in the future? Can the company help with the contribution of ArCon to implement those?

7.2.2 Eclipse Modeling Project

The Eclipse Modeling Project (EMP) contains several sub-projects to support Model-Driven development. It would be beneficial to make use of these projects and their offered functionality, and as result save development time by not developing the same functionality again in the ArCon project.

One of the interesting projects is UML2, that is an EMF-based implementation of the Unified Modeling Language 2.x OMG metamodel for the Eclipse platform. The current version of ArCon, v1.5, has its own representation of the UML elements in form of C++ classes. This representation is only a subset of the UML elements. The current transformation rules are thereby limited to only be checked on this subset of UML. As seen in the case study other elements are also subjects to be checked for architectural rules. All element types to be checked also need to have their representation within ArCon, and consequently new element types need to be added to ArCon's representation of UML. But creating a fully compliant version of UML and maintain it is a very large and unnecessary task for the ArCon project. Using the UML2 project, and its representation of UML, would save a lot of time, while also allowing new transformation rules to be developed. Other functionalities like XMI import/export are also provided within the UML2 project and could be used by the ArCon project. Using

UML2 also gives the advantage to work on the same representation as the modeling tools, such as Papyrus, within the Eclipse project. No adapter will be needed to convert UML elements between two different representations, if the applications would share the same representation.

Other projects, within Eclipse Modeling Project, may be of interest when the development of ArCon matures, or when new projects arise under the EMP flag.

7.2.3 Modeling Tool Integration

Returning a text message, containing any possible violations, as a result, works only as a basic approach to inform the user of any errors. More mature ways are needed to help and guide the user to actually tell where in the model the cause of a problem is located. This could be integrated into the modeling tools similar to how many IDE:s, for example Visual Studio, present errors from the code compilation. When given a compile error, the line where the error occurred is reported and the user can instantly jump to the location by double-clicking on the error message. In a similar way the user could be told more specifically where in the model the error is located, and the error message acts as a link to it. This may not be as straight forward as when compiling code since the UML model is visualized in several diagrams, and a construct can be visible in several diagrams, so it may not be obvious how and where to show the error.

The developers may also want to get live feedback on the architecture conformance during the development. The architectural rules may then also be checked while the developer modifies the system model, instead of just after the user explicitly called for a conformance check. This would be similar to how some code editors report code syntax errors while the developer writes the code, instead of just after the compilation.

7.2.4 Improved Mapping Techniques

As seen in the case-study, Section 6.2, the current use of stereotypes works in many cases but it is also limiting in other cases. More ways to map the rules than only using stereotypes are needed. As more described in the reflections in Section 6.3, some of the proposed techniques are being able to constrain general element types, meta classes, and element instances of a stereotype.

7.2.5 Extending Transformation Rules

The UML consists of several elements and diagrams but ArCon version 1.5 has its focus on the elements defined mostly within class diagrams. As seen in the case study other elements and diagrams are of interest to be checked, and with them other types of transformation rules may arise. How these

transformations will be defined and work is not obvious today and they will have to be evaluated just like the current rules.

An idea could be to create a framework to allow user-defined transformation rules, meaning they do not exist within ArCon's standard transformation regulations. The motivation would be that companies and stakeholders may have their own types of rules, which are specific for them, that they want to check. Allowing them to define and check those rules, without implementing user specific rules into ArCon, would be beneficial for them while also making sure ArCon remains a generic tool that can be used by everyone. If a user-defined rule is generic enough it can be subject to be added into ArCon's core transformation rules. It would then be of interest, for the community around ArCon, to actually take in new generic rules into ArCon's standard rules, to prevent everyone from having their own definition of transformation rules and resulting in nobody can interchange their models.

Defining Range of Types A specific suggestion, received during the case study presented in Section 6.2, was to add new data types that certain relations and attributes can have. The current UML syntax is not created to define a high-level description of a system that may contain very vague values and constrains. The example brought up during the case study was that the type of an attribute or parameter can only be defined specifically or totally unconstrained. But a possible case, stated by one of the testers, was that a type may actually be a range of possible types, for example it may be an integer or a string. From the architecture point of view it may not be specifically defined which one it should realize in the system model, and as a result the given type in the architectural rules model has to be able to allow those two types without allowing other types. This is not possible in the first version of ArCon where you can only give specific types or an unconstrained type, as in any type is allowed. A list type was suggested to define which possible types the attribute or parameter may have.

Multiplicity In current version of ArCon, arguments to operations and attributes for an element can have a wildcard '*%*' within their name. This wildcard allows the name, of the matching element in the system model, to have any type of string replacing the wildcard. For example the defined name '*%_Pkg*', in the architecture model, can be replaced with '*Sensor_Pkg*' in the system model. If the name, in the architecture model consist entirely of the wildcard '*%*' then it also has the meaning that the parameter or attribute may be repeated in any number, including zero [1]. This rule allows very few possibilities to constrain the system and may open up the system in way that is not intended. A possible suggestion is to use the multiplicity value, on the parameter and attribute, to define how many times the parameter and attribute may be repeated, using the same matching definition for name

[1] See transformation rule T12 and T13 on Table 1 for specification.

and type. Using the multiplicity separates the name definition from the definition of the allowed amount, while also allowing more specific constrains to be defined on the amount.

Forcing Values to be Defined Attributes and other constructs may have a default value that the developer can define. In the architectural model, the architect may not want to define what the default value should be, but only that the value should be explicitly defined in the system model. Thereby an option to say if an attribute, or other element, must have a default value in the system model could be of use in the future. The architect may also want to say that other values or settings should be explicitly defined by the developers, such as tagged values.

Missing Method to Restrict Values The T10 transformation rule, see Table 1, defines a rule on the relationship between two different stereotypes. But the rule only defines how many different element types, using the stereotype, that can have an association to the first element. There is no way to define the multiplicity on the allowed associations. An example can be seen in Figure 7.1 where the architectural rules model defines the following constrains: *a* `<<Data_Item>>` *class must have associations to two classes with the stereotype* `<<Sensor>>`. But there exists no rule to constrain the multiplicity on those associations. Even if the setting may not always be of interest to constrain, the option to constrain or not constrain should still exist.

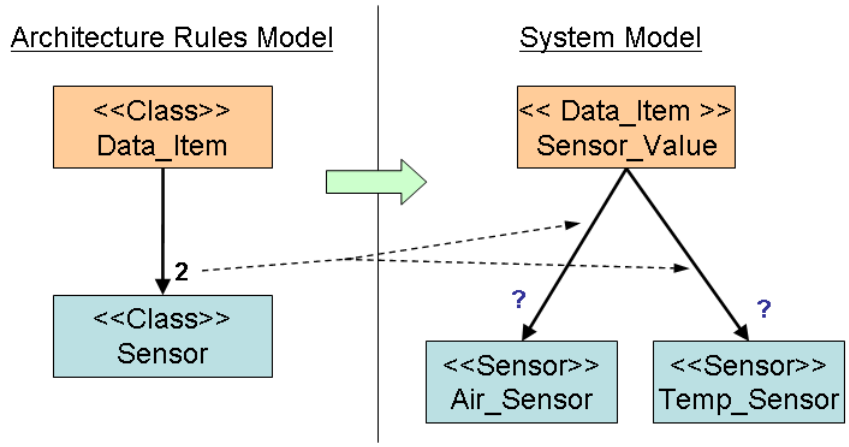


Figure 7.1: Illustration of transformation rule T10.

7.2.6 Formalization

The current transformation description, see Table 1, is vague and subject to interpretation by the reader. A formalized description of the rules is needed before the method can become a standard and be used more widely.

7.2.7 Tool Automation

As described in 3.3.1 it would be of interest to automatically generate the architectural profile, containing all the stereotypes used in the system model, from the architectural rules model. It seems like a possible future work to do when the transformation rules, that ArCon uses, have matured and are not subject to very much modification.

7.2.8 Profiling

One of the drawbacks with the current method to model the architectural rules, supported by ArCon, is that the constructs may not be easy to understand. Since the UML constructs have another meaning than the standard UML it can be confusing for the user. Also the UML elements are not created for this purpose, so there may also exist constructs that have no meaning at all, or the meaning is very vague in the context of the architectural meta model. As a consequence, I suggest extending and constraining the constructs with UML profiling. The idea is based on the way SysML is defined by UML profiling, and with that both extending and constraining the UML constructs. With the profiling it would be possible to have names and values so that the meaning of the model becomes more obvious. Constructs in the standard UML that have no meaning in the architectural meta model could also be removed, and new constructs can be added to better describe the architecture. Later on, if the method receives ground, modeling tools can add explicit support for the method in a similar way as many tools support SysML nowadays.

In the current transformation rule description, Table 1, the architectural rules model has to use some stereotypes to define some specific rules. At current state these would be created in a profile that the architectural model applies. This means the step towards profiling has indirectly already started but a lot more can be done to help the users understand and use the building stones of the architectural meta model.

7.3 Conclusion

Architectural design rules are seen as important, and at the same time they are complicated to define and verify. ArCon uses a proposed method to define and check the architectural design conformance of software systems. During this thesis, ArCon has been shown to have potential to define a wide

range of architectural rules. This range is however not without limitation, and there are rules that remain hard or impossible to define with the current semantics. Both the tool and method can be seen as early prototypes, and more work needs to be done before they can become fully mature. This thesis answered some questions and brought some improvements for ArCon and its method, while raising even more questions and casting light on the need for further research.

Bibliography

[1] C. Deiters, P. Dohrmann, S. Herold, and A. Rausch. Rule-based architectural compliance checks for enterprise architecture management. In *Enterprise Distributed Object Computing Conference, 2009. EDOC '09. IEEE International*, pages 183 –192, sept. 2009.

[2] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional; 1st edition, 1995. ISBN 0201633612.

[3] L Fuentes-Fernandez and A Vallecillo-Moreno. An introduction to uml profiles. *UPGRADE, The European Journal for the Informatics Professional*, 5(2):5 – 13, april 2004.

[4] A. Jansen and J. Bosch. Evaluation of tool support for architectural evolution. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 375 – 378, sept. 2004.

[5] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*, pages 109 –120, 2005.

[6] J. Knodel, D. Muthig, U. Haury, and G. Meier. Architecture compliance checking - experiences from successful technology transfer to industry. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 43 –52, april 2008.

[7] C.F.J. Lange, M.R.V. Chaudron, and J. Muskens. In practice: Uml software architecture and design description. *Software, IEEE*, 23(2):40 – 46, march-april 2006.

[8] Jinhua Li, Zhenbo Guo, Yun Zhao, Zhenhua Zhang, and Ruijuan Pang. Towards quantitative evaluation of uml based software architecture. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, volume 1, pages 663 –669, 30 2007-aug. 1 2007.

[9] A. Mattsson, B. Lundell, B. Lings, and B. Fitzgerald. Linking model-driven development and software architecture: A case study. *Software Engineering, IEEE Transactions on*, 35(1):83 –93, jan.-feb. 2009.

[10] A. Mattsson, B. Lundell, B. Lings, and B. Fitzgerald. An approach for modelling architectural design rules in uml and its application to embedded software. *ACM Transactions on Software Engineering Methodology (TOSEM)*, 21, jan.-feb. 2011.

[11] Anders Mattsson, Brian Fitzgerald, Björn Lundell, and Brian Lings. An approach for modeling architectural design rules in uml and its application to embedded software. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(2), march 2012.

[12] M.N. Miladi, M.H. Kacem, A. Boukhris, M. Jmaiel, and K. Drira. A uml rule-based approach for describing and checking dynamic software architectures. In *Computer Systems and Applications, 2008. AICCSA 2008. IEEE/ACS International Conference on*, pages 1107 –1114, 31 2008-april 4 2008.

[13] D. Perovich, M.C. Bastarrica, and C. Rojas. Model-driven approach to software architecture design. In *Sharing and Reusing Architectural Knowledge, 2009. SHARK '09. ICSE Workshop on*, pages 1 –8, may 2009.

[14] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley Professional, 2004. ISBN 0-321-11358-6.

Linköpings universitet
TEKNISKA HÖGSKOLAN

Avdelning, Institution
Division, Department

PELAB,
Dept. of Computer and Information Science
581 83 Linköping

Datum
Date

2012-06-12

Språk
Language
☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category
☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN

ISRN
LiU-Tek-Lic–2012:2012

Serietitel och serienummer **ISSN**
Title of series, numbering -

Linköping Studies in Science and Technology

Thesis No. LIU-IDA/LITH-EX-A-12/031-SE

URL för elektronisk version

http://www.ep.liu.se

Titel
Title

Architectural Rules Conformance - with ArCon and Open-Source Modeling Tools

Författare
Author

Emil Fridell

Sammanfattning
Abstract

In software development it is often crucial that the system implementation follows the architecture defined through design patterns and a constraint set. In Model-Driven development most artefacts are created using models, but the architectural design rules is one area where no standard to model the rules exists. ArCon, Architecture Conformance Checker, is a tool to check conformance of architectural design rules on a system model, defined in UML, that implements the system or application. The architectural design rules are defined in a UML model but with a specific meaning, different from standard UML, proposed by the authors of ArCon. Within this thesis ArCon was extended to be able to check models created by the Open-Source modeling tool Papyrus, and integrated as a plugin on the Eclipse platform. The method used by ArCon, to define architectural rules, was also given a short evaluation during the project to get a hint of its potential and future use. The case-study showed some problems and potential improvements of the used implementation of ArCon and its supported method.

Nyckelord
Keywords
Software Architecture, Model-Driven Development, UML, Architecture conformance, ArCon, Open-Source