UNIVERSITY OF TWENTE.

Change Management within SysML Requirements Models

David ten Hove

Master's thesis

University of Twente

Faculty of Electrical Engineering, Mathematics and Computer Science
Department of Computer Science
Software Engineering Group

June 24th 2010

Graduation committee:
University of Twente
- dr. ir. Klaas van den Berg
- dr. Ivan Kurtev
- Arda Goknil, Msc.

@Portunity
- Koos de Goede

Abstract

The need to properly manage change within software projects is very real. Not the least of the consequences of unmanaged or poorly managed change is decreased software maintainability. As changes in higher abstraction levels always propagate to the lower levels, it is important to manage change not only in code and design, but in requirements documents and requirements models as well.

We propose a method for managing changes within requirements. This method is based on keeping requirements models (expressed in SysML) in sync with what the stakeholders want, also known as the application domain. It focusses on determining which model elements are impacted, how they are impacted and how the impact has to be dealt with. Changes in the application domain are identified, a model element is designated to be affected and from this, the complete impact of the change on the model is calculated using the requirement relations as trace links. This calculation is performed using propagation rules, which are derived from the formalization of requirements, requirement relations and applicatoin domain changes we have created.

A running example is used to show the usage and practicality of the method. Some application domain changes of varying complexity are applied to an example requirements model to illustrate how the method performs in practice and how useful it can be. Furthermore, we provide tool support in the form of a prototype. This prototype, an extension of the commercial tool Blueprint, illustrates the user interactions in the process.

The method we propose has some limitations. First of all, it is partially automated. The experience and knowledge of a requirements engineer is still required for proper use. Second, it is based upon a very specific interpretation of the requirements relations in use. As there are many other viable interpretations possible, this method may not be applicable to every software project without adaptation. Apart from these limitations however, we believe the method does provide meaningful support to change management within software requirements models.

Acknowledgements

I would like to thank the people who have helped me during this research project.

First of all, my supervisors Arda, Klaas and Ivan for guiding me through the project by giving constructive feedback. The brainstorm sessions and discussions were very helpful and inspiring. Special thanks to Arda who, with his own work, provided me with a very good starting point for my research.

Also, I would like to thank Koos from @-portunity for giving me the possibility to apply my approach to an existing tool, rather than having to build my own from scratch.

My fellow students in room 5066 cannot go unmentioned here. They were always in for a laugh and willing to share their insights. Because of them, the Zilverling was never a dull place for me to work.

Finally, I want to thank my family. They never gave up on me and always supported me as best they could. I want to give special thanks to my parents, who always supported me and took care of me when my health was at its worst. Without their help, love and support I never would have been able to finish this work.

David ten Hove, June 2010, Enschede

Table of Contents

Abstract **iii**

Acknowledgements **v**

1 Introduction **1**
1.1 Context . . . 1
1.2 Problem statement . . . 3
1.3 Approach . . . 4
1.4 Overview . . . 5

2 Basic concepts **7**
2.1 MOF . . . 7
2.1.1 OMG four layered architecture . . . 7
2.1.2 Purpose of MOF . . . 8
2.2 UML . . . 8
2.2.1 Modeling using UML . . . 8
2.2.2 Customizing UML . . . 9
2.3 Using OCL with UML . . . 9
2.4 SysML . . . 10
2.4.1 SysML diagrams . . . 11
2.4.2 Modeling requirements in SysML . . . 11
2.5 Model Driven Engineering . . . 13
2.6 Change Management . . . 13
2.7 Summary . . . 13

3 Classification of changes **15**
3.1 Domain . . . 15
3.2 Domain change . . . 15
3.3 Model . . . 16
3.4 Model Change . . . 16
3.5 External inconsistency . . . 17
3.6 Internal Inconsistency . . . 17
3.7 Requirement parts and details . . . 17
3.8 Summary . . . 18

4 Impact Analysis Process **19**
4.1 Process . . . 19
4.2 Relations between terms . . . 20
4.3 External inconsistency propagation . . . 21
4.4 Summary . . . 24

5 Formalization of model elements and changes **25**
5.1 Introduction . . . 25
5.2 Relevant SysML model elements . . . 25
5.3 Formalizations . . . 26
5.3.1 Requirement . . . 26
5.3.2 Formalization of TracedTo . . . 27
5.3.3 Formalization of DerivedFrom . . . 28
5.3.4 Formalization of ComposedBy . . . 28
5.3.5 Formalization of CopyOf . . . 28
5.4 Domain change case analysis . . . 29
5.4.1 New requirement added . . . 29
5.4.2 Existing requirement removed . . . 29
5.4.3 Requirement made more specific . . . 30
5.4.4 Requirement made more abstract . . . 30
5.4.5 Part removed from requirement . . . 30
5.4.6 New part added to requirement . . . 30
5.5 Propagation rule derivation . . . 31
5.5.1 New part added to requirement . . . 31
5.5.2 Requirement made more abstract . . . 32
5.6 Discussion of formalization . . . 33
5.6.1 Using predicates and systems . . . 34
5.6.2 Using systems only . . . 34
5.7 Summary . . . 34

6 Example process usage **35**
6.1 Example Model . . . 35
6.2 Example changes . . . 36
6.2.1 Remove setting visibility of archived items . . . 37
6.2.2 Remove student team support . . . 38
6.2.3 Add support for removing grades . . . 39
6.3 Summary . . . 40

7 Tool Support **43**
7.1 Blueprint . . . 43
7.1.1 Blueprint capabilities . . . 43
7.1.2 Blueprint data structure . . . 44
7.2 Solution location . . . 44
7.3 Plugin development . . . 45
7.4 Plugin architecture . . . 46
7.4.1 AbstractChangeImpactAnalysisAction . . . 46
7.4.2 PartMadeMoreSpecific . . . 46
7.4.3 PartMadeMoreAbstract . . . 47
7.4.4 PartAddedToReq . . . 47
7.4.5 PartRemovedFromReq . . . 47
7.4.6 ReqRemoved . . . 47
7.4.7 ModelQuerier . . . 47
7.4.8 OCLModelQuerier . . . 47
7.4.9 RulesManager . . . 47
7.4.10 PropagationChooser . . . 48
7.4.11 StdInChooser . . . 48

7.4.12 ChangeImpactListener 48
7.4.13 StdOutListener 48
7.4.14 AbstractUMIModelCommandAction 48
7.5 Example usage 49
7.5.1 Example model 49
7.5.2 Usage 49
7.6 Summary 50

8 Conclusion **51**
8.1 Classification of changes 51
8.2 Determining and resolving impact 52
8.2.1 Process workings 53
8.2.2 Formalization and derivation 53
8.3 Tool support 54
8.4 Evaluation 54
8.4.1 Choice of metamodel 54
8.4.2 Formalization 54
8.4.3 Change Impact Analysis Process 55
8.4.4 Final remarks and future work 55

A Test Plan **57**
A.1 Basic tests 57
A.1.1 Testing domain change to external inconsistency to model change mappings 57
A.1.2 Testing the derivement relation 58
A.1.3 Testing composition relation 60
A.1.4 Testing copy relation 61

B Example model **63**
B.1 Example model 63

References **67**

1 Introduction

1.1 Context

Change management has always been a vital process within software development projects. Traditionally, it is performed more or less ad hoc. First, a change is requested, for example because a stakeholder presents a change in his or her interests. A software engineer uses his experience and knowledge of the software to then identify which software artifacts are affected by the change, how they are affected, and how they should be updated.

In smaller software projects, this approach may be prudent. However, when software projects get larger and more complex this may lead to serious issues. Studies have shown that when predicting change impact, software engineers tend to severely underestimate not only how many artifacts are impacted, but also how bad the impact will be for those artifacts [LS98].

[RI06] depicts the current status of requirement change management process models. It describes a multitude of activities within change management process such as planning for change, cost benefit analysis and submission of modification report. Their focus is on the organizational aspect of change management.

A lot of research has been performed in the field of change management, focussing on the effects of different methods (or lack thereof) and on improving these methods. For example, [JL04] identifies four common impact analysis strategies:

- Analyzing traceability or dependency information
- Utilizing slicing techniques
- Consulting design specifications and other documentation
- Interviewing knowledgeable developers

The first two of these methods are automatable processes, the last two are manual.

[BLOS06] defines an automated change impact analysis process in the context of UML models. It focusses on keeping several UML models of a single software system in sync with each other by propagating changes made to one model to related models. The process consists of four subprocesses:

- Verify the consistency of changed diagrams. In other words, check if the model is consistent with itself
- Automatically detect and classify changes. Classify according to a predetermined change taxonomy
- Perform impact analysis. Based on impact analysis rules, using the change taxonomy
- Prioritize the results of impact analysis.

Several other approaches have been developed. Although the previously mentioned approaches consider other software architects as well, most approaches are based on change impact analsysis to and from software code. These approaches certainly have their merits, but in larger software systems it is necessary to properly manage change in higher abstraction levels, such as the design and the requirements, as well.

One approach which focusses mostly on requirements is [GKvdB09]. This research defines its own metamodel of requirements and requirements relations with a well-defined mathematical basis. They apply a distinction between changes based on the notion of the application domain[1]. The domain is that which the stakeholders have an interest in, and that which the model has to represent. The types of changes are:

- A change which is necessitated by a change to the domain. These affect the functional and non-functional properties of the required system.
- A change which is not related to a change in the domain. These do not affect the functional or non-functional properties of the required system. They are usually refactoring changes.

When a change is made of either type, the requirements relations are used as traces to determine if the change needs to be propagated to other model elements. This propagation is performed by use of propagation rules, which are based on the mathematical definition of requirement, requirement relations and changes. This work has provided us with a good starting point for our research and excellent comparison material, even though there are several elements which make the two researches very much distinct.

The main distinctions between our work and [GKvdB09] are that we use a different modeling basis and that we do not support managing changes which do not stem from changes in the domain.

As in [GKvdB09], we consider only change management within requirements models, as shown in figure 1.1. The boxes in this picture (apart from the domain) show the levels of software development with artifacts. The artifacts in the requirements level can be extracted from the domain. The artifacts in the high level design can be derived from the requirements, and so on. When the

[1] We refer to the application domain simply as the domain

requirements have been derived from the domain and later on a change occurs in the domain, the requirements may need updating. This is what we intend to address and it is shown in the figure by the solid line. Note that this kind of propagation holds for any two layers, and that this is also transitive.

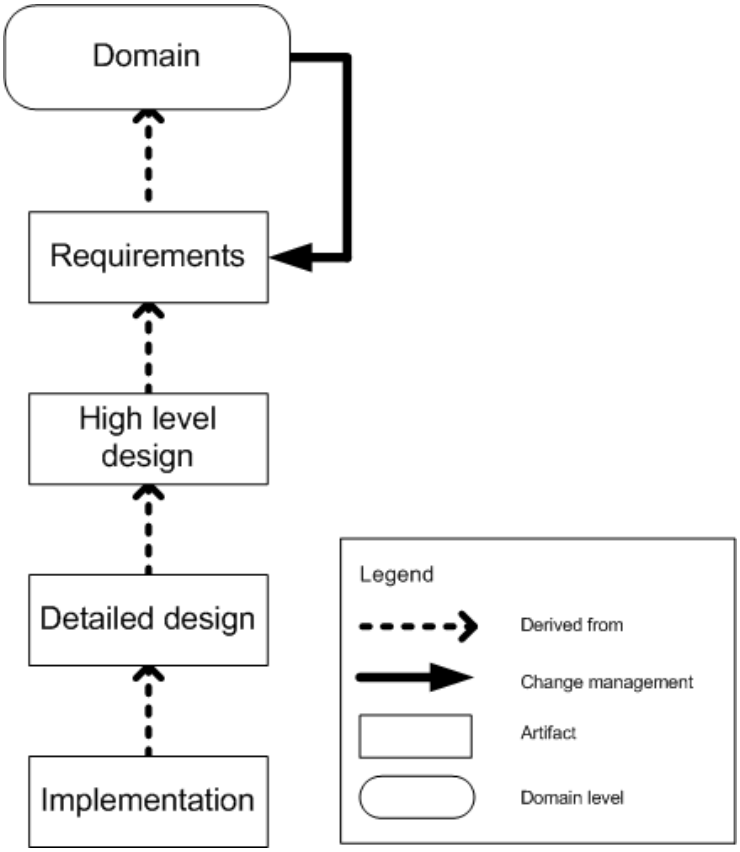

Figure 1.1: Abstraction levels in software development

Current tool support for change management within requirements models is limited. One tool which is widely used is Rational RequisitePro, created by IBM [Req]. Amongst other things, this tool allows users to specify requirements and to specify which requirements are related to one another. It is not possible however, to specify exactly how requirements are related, which severely limits the change management possibilities.

1.2 Problem statement

The main problem we deal with in this paper lies in change management within software projects. There are many aspects to change management, including how to perform it organizationally and applying cost benefit calculations. We consider only the technical side of the problem, i.e. the effect of the change on the software and how to handle it within the software.

Change management can be performed on different abstraction levels, such as the design or the code. The effect of a change can easily span over several levels, and in perfect change management all levels would be considered. However, since this would be a monumental task, we consider only the levels of domain and requirements. There are several ways in which the requirements can be represented. We use a subset of the SysML metamodel developed by OMG, because it contains a number of predefined requirements relations. See also [OMG07a].

The largest issue with the current approaches, is that the relations lack semantics. This causes the impact analysis to be very imprecise in that there will almost always be false positives. In many cases, a change to a single requirement will affect a very large part, if not all of the model. This is known as the impact explosion.

Our main research question is the following: **Given a change in the domain, how does the SysML requirements model have to be updated in order to correctly reflect the domain?**

In order to answer this question, we first have to determine the impact of the change. This involves determining which model elements are impacted, and how.

Determining which elements are impacted is done by examining the type of the domain change and the relations between the requirements in the model. So, in order to perform this step correctly, we need to classify the domain changes and formalize them along with the requirement and the requirement relations.

Determining how elements are impacted also involves determining the kinds of impact a domain change can have on the model. These impacts will also need to be classified. As a final step, the model needs to be updated according to the information gathered in the previous two steps. In order to update the model, we need to classify model changes and relate them to types of impact.

Our overall goal is to provide support for change impact analysis within software projects. The research questions are divided into three categories:

1 Classification

- 1.1 What are the types of domain changes
- 1.2 What are the types of impact of domain changes on model elements

2 Determining and resolving impact

- 2.1 How can we determine which model elements are impacted
- 2.2 How can we determine the type of impact on these elements
- 2.3 How can we solve the impact of domain changes on model elements

3 Tool support

- 3.1 How can we use tools to support change management

1.3 Approach

We start our work by doing literature research in the field. This includes other change management research and requirements management research. Most notably, we will closely examine [GKvdB09], but also other existing requirements

management methods. We will also look at existing requirements management tools. A running example will be used to determine the viability of the process and a prototype will be constructed to examine tool support possibilities.

1.4 Overview

The document is divided into the following parts:

- Chapter 1. Introduction
- Chapter 2. Basic concepts.
- Chapter 3. Definitions and classifications.
- Chapter 4 and 5. Impact analysis process and mathematical foundation.
- Chapter 6. Example usage of impact analysis process.
- Chapter 7. Tool support.
- Chapter 8. Evaluation and conclusion
- Appendices. Test plan and example model.

2 Basic concepts

This chapter describes some the basic concepts we use in our thesis. These include various modeling standards, model driven engineering and change management.

2.1 MOF

The MetaObject Facility (or MOF) is a standard adopted by OMG which defines a metamodel that is currently being employed by a number of widely used technologies, such as UML and SysML, which will both be described later in this chapter. It is defined in [OMG06].

2.1.1 OMG four layered architecture

MOF is at the top of the OMG four-layered architecture, shown in figure 2.1.

- `M0` is the data layer, such as the program being executed.
- `M1` is the model layer, which contains models as built by system engineers. It is this model the data in layer M0 conforms to.
- `M2` is the metamodel layer, which contains metamodels such as UML and SysML. A systems engineer applies this metamodel to create models of layer M1.
- `M3` is the meta-metamodel layer, which contains the metamodels of the metamodels, including MOF. In the OMG four layered architecture, there is no higher level. The metamodel of MOF is MOF. That is, MOF is defined in itself.

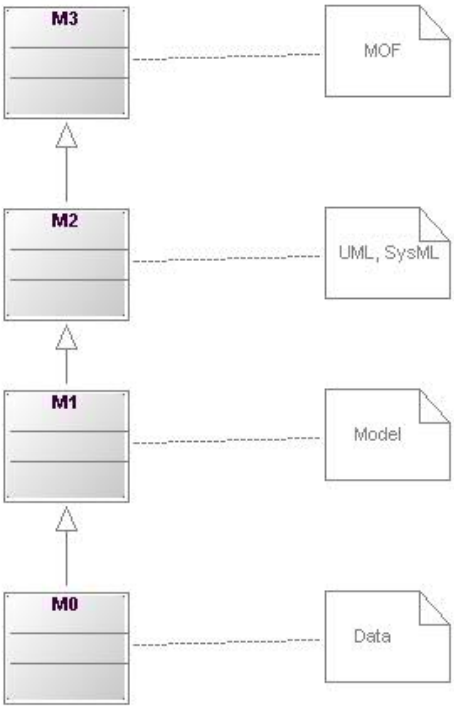

Figure 2.1: OMG architectural layers. Based on [OMG02]

2.1.2 Purpose of MOF

The main purpose of MOF is to provide interoperability of models. A model which conforms to MOF can be imported, exported and edited by any tool with the possibilities. If no such higher-level metamodel would exist, tools would have to be customized for every (meta)model it would want to work with. However, it is not only tools which benefit from MOF, but also technologies. The Object Constraint Language for example, which is explained later in this chapter, allows developers to apply constraints to any model which conforms to MOF.

2.2 UML

2.2.1 Modeling using UML

The Unified Modeling Language (UML) is a language for specifying, constructing and documenting object-oriented software systems and is defined in [OMG07b]. It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all implementation platforms, such as J2EE and .NET. UML is also a visual language, which means it is specified in diagrams. UML specifies several diagram types, the most commonly used including:

- `Class diagrams`. Describe the static structure of a system by showing

the classes, attributes, operations and the relations between classes.

- `Use case diagrams`. Describe the functionality of a system by showing the actors, the goals and the relations between these.
- `Sequence diagrams`. Describe the workings of a system by showing the objects and the messages exchanged between them.

These diagrams, among with many others in UML, are used by software developers throughout the world to specify and document their software systems. As stated before, UML uses MOF as its metamodel.

2.2.2 Customizing UML

In many cases, it is desirable to customize UML to a certain domain. Customization is possible using profiles, which define an extension to any reference metamodel (such as UML). The only requirement for the reference metamodel is that it conforms to MOF. Each profile contains any number of stereotypes that extend an already existing element of the metamodel, e.g. a metaclass or an actor. Stereotypes have meta-attributes called tagged values. When creating a model which conforms to a metamodel with a specific profile applied to it, the elements with custom stereotypes have to have a concrete value for each tagged value of that stereotype. Figure 2.2 shows how MOF, UML and UML profiles relate to each other.

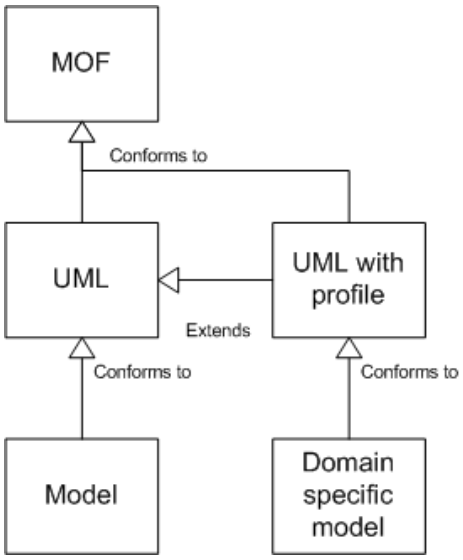

Figure 2.2: MOF, UML and UML profiles

2.3 Using OCL with UML

The Object Constraint Language (OCL) is a formal language first used to describe expressions on UML models and is defined in [OMG03]. It has been

adapted by OMG so that it can now be applied to any MOF-based model. These expressions typically specify invariant conditions that have to hold for the system being modeled, but can also be used to define pre- and postconditions. OCL is necessary for these constraints as there is often no other way in these models to describe them formally. Of course, natural language can be applied, but this requires the constraints to be checked manually, as opposed to automatically.

The constraints can range from simple to very complex. As a simple example, consider a class Person with a single attribute: age. A sensible constraint would be to ensure that the age of a person is always greater than or equal to zero. In OCL, it could look like this:

```
context Person inv ageConstraint:
self.age >= 0
```

The constraint first specifies the context as the class int Person. Next, the type is set: **inv** for invariant. Other possible types are **pre** and **post**. It then names itself ageConstraint. The second line specifies that the attribute reached with self.age has to have a value higher than or equal to zero. Note that self. may be omitted in this case.

2.4 SysML

SysML is a general-purpose modeling language for systems engineering that is designed to provide simple but powerful constructs for modeling a wide range of systems engineering problems and is defined in [OMG07a]. It supports modeling and specifying requirements, structure, behavior, allocations and constraints of systems. In the same way the Object Management Group has introduced UML in an attempt to standardize software modeling, SysML is intended to standardize modeling and engineering of almost any kind of system. SysML reuses a subset of UML 2, called Uml4SysML. Figure 2.3 shows how Uml and SysML relate to each other.

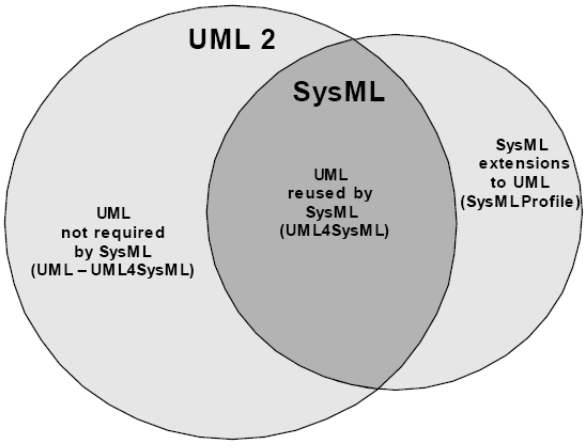

Figure 2.3: UML and SysML. [OMG07a]

The region marked "SysML Extensions to UML" indicates the new modeling constructs for SysML which do not have a counterpart in UML, or replace UML constructs. Note that, as the figure shows, there is also a part of UML which is not used by SysML. The fact that SysML reuses a large part of UML makes it easier for engineers who have experience with UML to adopt the new language.

2.4.1 SysML diagrams

SysML is a graphical language, like UML. It reuses a number of the diagrams specified in UML, but also introduces two new ones. Figure 2.4 shows the diagrams in SysML.

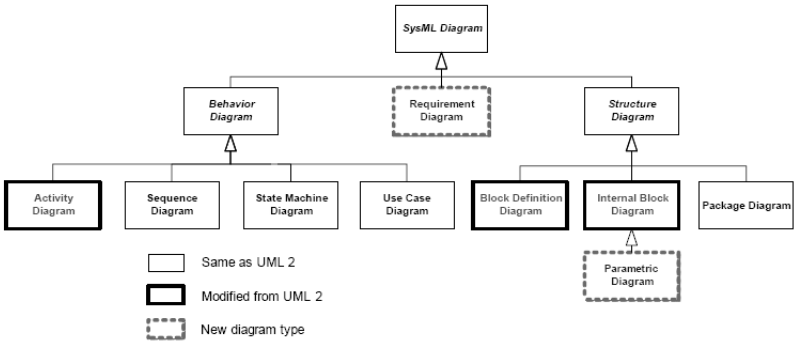

Figure 2.4: Diagrams in SysML. [OMG07a]

The two diagrams introduced are requirement and parametric. A parametric diagram is defined as a restricted form of internal block diagram. I t describes constraints, constraint parameters and constraint blocks. These constraints limit the physical properties of a system and can be used to identify critical performance parameters and their relations to other parameters. Parametric diagrams can be used to support trade-off analysis, since the factors in the applicable equations are objectively and formally presented.

A requirement diagram displays requirements, packages, other classifiers, test cases and rationale, along with their relations. It is used to structure requirements and to link them to other design elements within SysML.

2.4.2 Modeling requirements in SysML

SysML defines a requirement as "a capability or condition that has to (or should) be satisfied". A requirement may specify a function that a system has to perform or a performance condition a system has to achieve. They are modeled as a stereotype of UML Class subject to a set of constraints. A standard requirement includes properties to specify its unique identifier and text requirement. There are several relations which can be used by system modelers to specify how requirements relate to each other, but also to other model elements. In simple systems, these relations may be implicitly clear, but they are very important in modeling the intricacies of larger, more complex systems. The relation types for requirements in SysML are:

- `Composite` relations allow a requirement to contain sub-requirements
- `Copy` relations define links between a master requirement and a copy of it, known as a slave requirement
- `Derive` relations allow one requirement to be derived from another
- `Satisfy` relations define a model element as satisfying a requirement
- `Verify` relations define a test case as verifying a requirement
- `Refine` relations define a model element or set thereof as refining a requirement
- `Trace` relations define a generic relation between a requirement and any other model element

A `composite` requirement can contain subrequirements to specify hierarchy. This is done using the UML namespace containment mechanism. The relation allows complex requirements to be decomposed into simpler ones to make things more concrete, or to apply abstraction to the model. Of course, subrequirements themselves may be composed of additional child requirements, and so on.

`Copy` relations specify slave and master requirements. SysML defines slave requirements as requirements whose text property is a read-only copy of the text property of its master requirement, where the master requirement may be declared in a completely different namespace. This means that the master requirement could be defined in a different project altogether, and without this relation it may be impossible to refer to it. The slave requirements cannot be changed, but when the master requirement is changed, so are all its slaves. This has been included in the SysML specifications in order to provide better support for requirement reuse, something which the designers of SysML discovered to be very important.

Another relation specified is the `derive` requirement, which relates a derived requirement to its source requirement. In other words, a client requirement can be derived from the supplier requirement. This typically involves analysis to determine the multiple derived requirements that support a source requirement. This relation is commonly used to define multiple layers of a system hierarchy.

The `satisfy` relation describes how a design or implementation model satisfies one or more requirements. The satisfying element can be almost anything in SysML.

The `verify` relation defines how a test case verifies a requirement. Test cases in SysML are intended to be used as a general mechanism to represent any of the standard verification methods for inspection, analysis, demonstration or test. The verdict of a test case can be used to represent the verification result.

Using the `refine` relation it is possible to describe how a model element can be applied to further refine a requirement, where the model element can be almost anything in SysML.

Additionally, SysML supports `trace` requirement relations, which provide a general-purpose link between a requirement and any other model element. Since this is intented to be a generic relation, it is recommended that it not be used in conjunction with the other requirements relations described above.

2.5 Model Driven Engineering

Model driven engineering means modeling not with standard classes and interfaces such as those found in UML, but with the components relevant to a certain domain. For example, one could model students, teachers and courses. In order to do this, a suitable Domain Specific Language needs to be defined.

There is a number of advantages to applying this kind of modeling. With conventional modeling, the modelers had to be experts both in the field of technology and in the field of the specific domain. Possibly the biggest advantage is that this is no longer necessary. In the ideal case, an expert on a domain who has no technical knowledge whatsoever can model the things he knows, and the engineers can then implement the software system based on this model. Currently, there is a gap between the state of the practice and the states of the art regarding model driven engineering.

2.6 Change Management

Managing changes in the context of software engineering is a prerequisite for high quality software management. This includes changes at every level of the development, whether it be the requirements, the architectural design or the implementation. In general, the higher the level at which change occurs, the bigger the impact. One aspect of change management is change impact analysis. This section describes change impact analysis at the design level. For simple, small models, change impact analysis occurs by user intuition and experience. However, in larger, more complex models, the need for an automated change impact analysis is very real.

[KS98] describes a change management process. A simple overview of this process as described there is shown in figure 2.5. This assignment fits in the general process as three elements: "Find affected requirements", "Find dependent requirements", and "Propose changes".

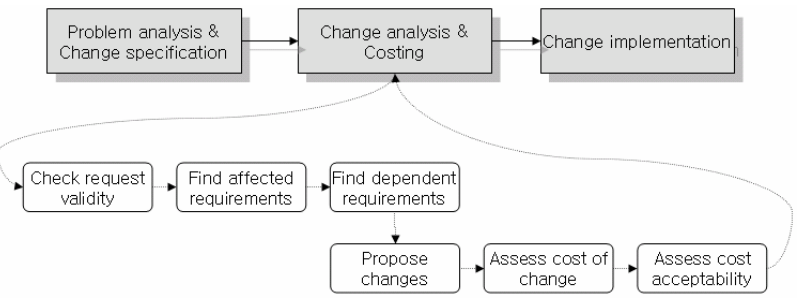

Figure 2.5: Change management process. [KS98]

2.7 Summary

This chapter described some of the standards relevant to our work, including MOF, UML, SysML and OCL. MOF defines a metamodel which is currently

being used by several other standards, including UML and SysML. UML is a visual language for specifying, constructing and documenting object-oriented software systems. OCL makes it possible to define expressions on MOF-based models. SysML is a general-purpose modeling language for systems engineering that is designed to provide simple but powerful constructs for modeling a wide range of system engineering problems.

We also described model driven engineering, which is to model not with standard classes and interfaces such as those found in UML, but with the components relevant to a certain domain. Change management in this context is the process of identifying change, finding which parts of a model are affected and how, and finally proposing model changes.

3 Classification of changes

In this chapter we introduce the terms we use in the change impact analysis process and classify domain changes, model changes and external inconsistencies.

3.1 Domain

Before we can begin to model anything, we need a name for it: The application domain (which we refer to simply as the domain). The concept of domain is described in [vL09], although it is not labeled there as such. The domain is what the stakeholders want to be modelled, and possibly implemented as well. These stakeholders have their own requirements, each specifying a functional or non-functional need.

Domain: The part of reality that needs to be modelled, viewed through the requirements it sets for the resulting system.

Changes in the domain can occur in several ways. For example, a stakeholders mind could change or the environment could change. Changes to the environment can be economical, jurisdictional etc.

3.2 Domain change

Below is a classification of types of change to a domain.

1 New requirement added

2 Existing requirement removed

3 Requirement made more specific

4 Requirement made more abstract

5 Part removed from requirement

6 New part added to requirement

This classification is based on a case analysis, described in section 5.4. It is not elaborated upon any further in this section, as it is entirely based on mathematical formalization.

3.3 Model

The term model has an intuitive meaning to most software engineers. We base our definition from [SSB93] and [FHL+98]. It is that which reflects the reality (or in this case, domain) as well as possible, or that which needs to be built. There are three properties of the concept of a model which are very important. The first is that it is an abstraction of reality. Certain details are deliberately omitted. The second is that a model serves a purpose. The purpose of a model is the guiding principle in deciding which parts of reality to include. The third property is that a model is always associated to a domain. Models are expressed in modeling languages.

Model: A model represents a part of the reality called the domain and is expressed in a modeling language. A model provides knowledge for a certain purpose that can be interpreted in terms of the domain.

3.4 Model Change

Below is a classification of types of change to a model.

1 Requirement is added

2 Requirement is removed

3 Detail added to requirement description

4 Detail removed from requirement description

5 Part removed from requirement description

6 Part added to requirement description

7 Relation is removed

8 Relation is added

As will become clear further on in this chapter, the last two model changes are never applied in the process.

3.5 External inconsistency

The change impact analysis approach is focused on keeping the model synchronized with the domain. External inconsistencies define a difference between the model and the domain. They can be caused by a domain change. If so, the type of domain change determines the type of external inconsistency. They are also always related to one or more model elements, or the lack of thereof.

External Inconsistency: An external inconsistency defines which model elements (or lack thereof) in a model do not conform to the domain.

Below is a classification of external inconsistencies, based on the classification of domain changes.

1 Requirement in the domain, but absent in the model

2 Requirement not in the domain, but present in the model

3 Requirement in the model less specific than in the domain

4 Requirement in the model more specific than in the domain

5 Requirement in the model has more parts than in the domain

6 Requirement in the model has less parts than in the domain

3.6 Internal Inconsistency

While external inconsistencies define a difference between the model and the domain, internal inconsistencies define a conflict within the model itself. Internal inconsistencies are caused by applying relations in a way that violates its formalization. Chapter 5 contains this formalization.

Internal inconsistency: A violation of relation constraints within a model.

3.7 Requirement parts and details

SysML defines a requirement as "a capability or condition that has to (or should) be satisfied". We define a requirement consisting of several parts, which serve as predicates of an implementation. That is, for every possible implementation, the predicate can be evaluated to true or false. The number of parts a requirement has is not limited, but it has to be at least one. A part can have any number of details applied to it, which refine the part. In other words, adding details to a part makes it more restrictive and decreases the number of implementations which satisfy the part. Details can also have additional details applied to themselves, but this cannot be cyclical. A detail is always associated to a single part or a single detail, and never has value on its own. Consider, for example, the requirement description "The system shall perform function X in Y seconds". "The system shall perform function X" is a part, whereas "in Y seconds" is a detail of that part. Note that it is not necessary for requirements engineers to decompose their requirements models before being able to apply this approach, it is merely the basis for it. For a more formal description of parts and details, see section 5.3.

3.8 Summary

In this chapter we defined several terms. We defined the domain as that in which the stakeholders have an interest in and therefor has to be modeled. We defined the model as that which reflects the domain. External inconsistencies are defined as inconsistencies between the model and the domain and internal inconsistencies as inconsistencies within the model itself.

Requirements are defined as sets of parts, which can have details applied to them. For any given implementation, the parts of a requirement can be evaluated to true or false, taking the details in consideration. Details limit the implementations which satisfy a part and do not have any meaning on their own. We also classified the following: Domain change, model change and external inconsistency.

Currently, we consider adding and removing elements from a requirement, but not changing a requirement in one step. Changing a requirement in this fashion can be seen as first adding and then removing elements. Additionally, we only consider adding and removing a single part or detail at a time. For the current time, it is sufficient, albeit tedious.

4 Impact Analysis Process

This chapter describes the change impact analysis process. It shows the relevant SysML relations and the tables containing the rules for external inconsistency propagation and the rules for mapping domain changes to external inconsistencies to model changes.

4.1 Process

The change impact analysis process starts with the identification of a domain change, which should be performed by a requirements engineer. The requirements engineer then splits the domain change up into one or more primitive domain changes. For each of these primitive domain changes, the engineer identifies an element in the model which is now inconsistent with the domain, and also how it is inconsistent. This is also the reason why the domain change first has to be split up into primitive domain changes. Without doing so, it may not be possible to pin it down to one model element. After this step the semi-automatic process starts. Using the relations in the model, the process can semi-automatically determine which other model elements are also externally inconsistent. Finding all external inconsistencies is done recursively. After all have been identified, the process proposes model changes to eliminate them. Figure 4.1 shows the entire process schematically. Identifying the domain change and splitting it into one or more primitive domain changes is done entirely by the requirements engineer, as well as identifying a model element relevant to the inconsistency. Recursively finding the other external inconsistency is done partly automatic. The system identifies possible external inconsistency propagation scenarios. The requirements engineer will then have to select the correct one. This could be fully automated if the requirements were fully formalized. However, fully formalizing requirements is extremely hard to do and demanding so would raise questions regarding the cost and benefits of this approach. Mapping the external inconsistencies to proposed model changes is entirely automated.

The exact implementation of the model changes is undefined here. It may be left completely to the user, but parts of it may also be automated.

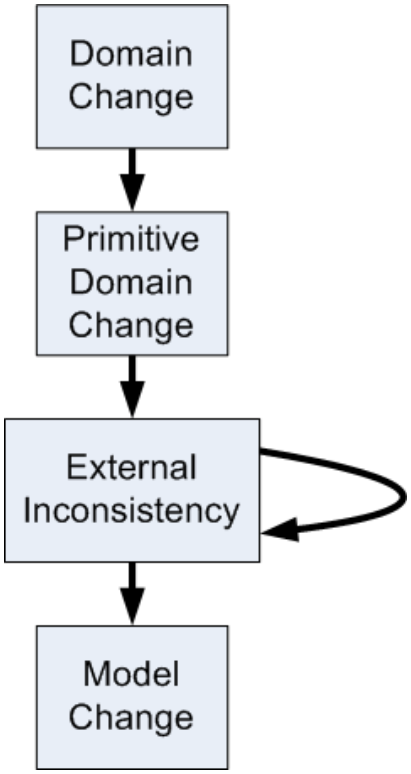

Figure 4.1: Change impact analysis process

The change impact analysis process described further down in this chapter relies on a critical assumption: The model is internally consistent. The results of the process if performed on a model which is internally inconsistent are undefined.

4.2 Relations between terms

Table 4.1 shows how the domain changes are mapped to external inconsistencies, and in turn, how these are mapped to model changes. These mappings are performed automatically by the process.

Domain change	External inconsistency	Model change
New requirement added	Requirement in the domain, but absent in the model	Requirement is added
Existing requirement removed	Requirement not in the domain, but present in the model	Requirement is removed
Requirement made more specific	Requirement in the model less specific than in the domain	Detail added to requirement
Requirement made more abstract	Requirement in the model more specific than in the domain	Detail removed from requirement
Part removed from requirement	Requirement in the model has more parts than in the domain	Part removed from requirement
New part added to requirement	Requirement in the model has less parts than in the domain	Part added to requirement

Table 4.1: Relation between domain changes, external inconsistencies and model changes

4.3 External inconsistency propagation

In order to perform external inconsistency propagation, we need rules which apply to specific relation types and external inconsistencies. For the moment, we consider only relation types between requirements. SysML also provides several relation types between requirements and other model elements, such as test cases, but these are not considered here. There are four relevant relations:

- `Trace` relations define a generic relation between requirements. For readability, this relation is called `traced to` in this paper
- `Composite` relations allow a requirement to contain sub-requirements. For readability, this relation is called `composed by` in this paper
- `Copy` relations define links between slave and master requirements. For readability, this relation is called `copy of` in this paper
- `Derive` relations allow one requirement to be derived from another. For readability, this relation is called `derived from` in this paper

Tables 4.2 and 4.3 show the external inconsistency propagation rules. They are derived in chapter 5. The external inconsistencies in the first column are the same as specified in table 4.1. The cells themselves contain the propagation rules for each scenario. Some external inconsistencies do not propagate in certain scenarios, while others have multiple propagation possibilities. The correct propagation has to be selected manually.

The `copy of` relation in SysML is not included in the tables for readability purposes. External inconsistencies which apply to a requirement which copies or is copied by another requirement, are automatically propagated to the other end of the relation.

Another relevant relation in SysML which is also not shown here is the `traced to` relation. This relation is specified in SysML as a generic relation with no semantical definition. As a result, no assumptions can be made on the propagation of external inconsistencies regarding the source or target of this type of relation. It is recommended for requirements engineers to avoid the use of this relation if at all possible (the SysML specifications disencourage using it in combination with other relation types). This approach does not propagate any external inconsistencies between a requirement and another which is traced to or from it.

Table 4.2 shows the propagation of external inconsistencies with regard to the `derived from` relation. A requirement can be `derived from` one requirement, or more. The table makes a distinction between these two cases, as the propagation rules are different.

Table 4.3 shows the external inconsistency propagation with regard to the `composed by` relation. Again, a distinction is made between scenario's in which a requirement is composed by one other requirement, and scenario's in which a requirement is composed by multiple other requirements.

Scenario	R_1 **derived from** R_2	R_1 **derived from** R_2 **through** R_n $k \in 2 \dots n$
External inconsistency		
R_1 not in domain	R_2 not in domain	For each R_k: not in domain
R_2 not in domain	R_1 not in domain	Part of R_1 not in domain *or* R_1 not in domain
R_1 less specific than in domain	No propagation	No propagation
R_2 less specific than in domain	R_1 less specific than in domain	R_1 less specific than in domain
R_1 more specific than in domain	R_2 more specific than in domain *or* no propagation[1]	For each R_k: more specific than in domain *or* no propagation[1]
R_2 more specific than in domain	R_1 more specific than in domain	R_1 more specific than in domain
R_1 has more parts than in domain	R_2 has more parts than in domain[2]	For each R_k: not in domain *or* part not in domain[2] *or* no propagation[3]
R_2 has more parts than in domain	R_1 has more parts than in domain[2]	R_1 has more parts than in domain[2]
R_1 has less parts than in domain	R_2 has less parts than in domain	For each R_k: less parts than in domain *or* no propagation[3]
R_2 has less parts than in domain	R_1 has less parts than in domain	R_1 has less parts than in domain
R_4 in domain, not in model	No propagation	No propagation

[1] In this scenario, a detail that made R_1 a derivement from the other requirement(s) has been removed from the domain. If there are no other such details, the relation or relations are no longer valid and should be removed

[2] In this scenario, it is possible that the last part which R_1 was deriving from the other requirement has been removed. If so, the relation is no longer valid and should be removed

[3] In this scenario, at least one external inconsistency has to be propagated to at least one other requirement

Table 4.2: External inconsistency propagation rules for the Derived From relation

Scenario	R_1 **composed by** R_2	R_1 **composed by** R_2 **through** R_n $k \in 2 \dots n$
R_1 not in domain	R_2 not in domain	For each R_k: not in domain
R_2 not in domain	R_1 has more parts than in domain	R_1 has more parts than in domain
R_1 less specific than in domain	R_2 less specific than in domain *or* no propagation	For each R_k: less specific than in domain *or* no propagation
R_2 less specific than in domain	R_1 less specific than in domain	R_1 less specific than in domain
R_1 more specific than in domain	R_2 more specific than in domain *or* no propagation	For each R_k: more specific than in domain *or* no propagation
R_2 more specific than in domain	R_1 more specific than in domain	R_1 more specific than in domain
R_1 has more parts than in domain	R_2 not in domain *or* part of R_2 not in domain *or* no propagation[1]	For each R_k: not in domain *or* part not in domain *or* no propagation[1]
R_2 has more parts than in domain	R_1 has more parts than in domain	R_1 has more parts than in domain
R_1 has less parts than in domain	No propagation	No propagation
R_2 has less parts than in domain	R_1 has less parts than in domain	R_1 has less parts than in domain
R_4 in domain, not in model	No propagation	No propagation

[1] In this scenario, a part has been removed from R_1 which was not in the other requirement. It is possible that the requirements are now copies and that the relation should be removed.

Table 4.3: External inconsistency propagation rules for the Composed By relation

4.4 Summary

This chapter describes the change impact analysis process and the mappings from domain changes to external inconsistencies to model changes. Lastly, the external inconsistency propagation rules are provided.

Some scenarios of external inconsistency propagation result in two requirements possibly becoming identical. The process could better support this. For example by proposing a model change removing the relation or by merging the requirements. Since the definition of the relations in the SysML specifications is ambiguous, this entire process is based on one specific interpretation and formalization of these definitions. Other interpretations of these relations would lead to entirely different external inconsistency propagation rules.

5 Formalization of model elements and changes

This chapter contains the formalization of the relevant SysML model elements and of the domain changes. We also use these formalizations to derive the external inconsistency rules here.

5.1 Introduction

This chapter first describes the relevant SysML model elements. Next, all of these model elements and the domain changes are mathematically formalized. Using these formalizations, the external inconsistency propagation rules are derived. Finally, some alternatives for the formalizaton are discussed.

5.2 Relevant SysML model elements

As stated in section 4.3, the relevant model elements are `Requirement`, `Trace`, and all subclasses of `Trace`. The other relations, such as `Satisfy` and `Refine` are not considered as of now. `Trace` and its subclasses are:

- `Trace` relations define a generic relation between requirements. For readability, this relation is called `traced to` in this paper
- `Composite` relations allow a requirement to contain sub-requirements. For readability, this relation is called `composed by` in this paper
- `Copy` relations define links between slave and master requirements. For readability, this relation is called `copy of` in this paper
- `Derive` relations allow one requirement to be derived from another. For readability, this relation is called `derived from` in this paper

5.3 Formalizations

5.3.1 Requirement

A `Requirement` R consists of a set of predicates P. Each of these predicates evaluate to true or false for any given implementation, or system. They have a textual description and a set of details D applied to them which, in turn, can have details applied to them as well. The details themselves are also textually described, but are meaningless when not associated to a predicate or another detail. Formally, this is denoted as:

- $R =< P >$
- $P = \{p_1\{< D_1 >\}, p_2\{< D_2 >\}, \ldots, p_n\{< D_n >\}\} \mid n \geq 1$
- $D = \{d_1\{< D_1 >\}, d_2\{< D_2 >\}, \ldots, d_m\{< D_m >\}\} \mid m \geq 0$

Figure 5.1 shows this formalization in a diagram.

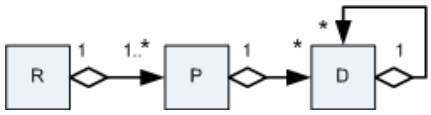

Figure 5.1: Formalization data model

The P in the figure denotes the individual predicates in the requirements, of which a requirement has any number, but at least one. The D in the figure denotes the individual details in either another detail or a predicate.

Sets of predicates cannot be empty, sets of details can. Details cannot be associated in a cyclical manner. In order to formalize the relation types, we need operators on requirements, predicates and details.

- Two predicates are equal to each other with regard to the = operator if and only if their textual descriptions are equal. Details are not considered here
- Two details are equal to each other with regard to the = operator if and only if their textual descriptions are equal and they are applied to two predicates which are also equal to each other with regard to the = operator or two details which are also equal to each other with regard to the = operator
- Consider a predicate p and a detail d. The detail d is associated to the predicate p if, and only if, it is directly applied to it, or if it is associated to another detail which is associated to the predicate p. "Detail d is associated to p" is denoted as $d \in p$.
- Consider a predicate p and two details d_1 and d_2, with $d_1 \in p$ and $d_2 \in p$. The detail d_1 is associated to the detail d_2 if, and only if, it is directly applied to it, or if it is associated to another detail which is associated to the detail d_2. "Detail d_1 is associated to d_2" is denoted as $d_1 \in d_2$.

- When evaluating a predicate p in a system S, $p(S)$ denotes that p with all details $d \in p$ is evaluated to true in system S.

For the following operator definitions, let R_1 and R_2 be requirements with sets of predicates P_1 and P_2 respectively.

- $|P_1|$ denotes the number of predicates in P_1

- $P_1 = P_2 \Leftrightarrow |P_1| = |P_2| \wedge$
  $\forall p_i \in P_1 : [\exists p_j \in P_2 : [p_i = p_j]] \wedge$
  $\forall p_i \in P_2 : [\exists p_j \in P_1 : [p_i = p_j]]$

- $P_1 \subset P_2 \Leftrightarrow |P_1| < |P_2| \wedge \forall p_i \in P_1 : [\exists p_j \in P_2 : [p_i = p_j]]$

- $P_1 \subseteq P_2 \Leftrightarrow P_1 = P_2 \vee P_1 \subset P_2$

- $R_1 = R_2 \Leftrightarrow P_1 = P_2 \wedge$
  $\forall p_i \in P_1 : [\forall p_j \in P_2 : [p_i = p_j \longrightarrow \forall d_k \in p_i : [d_k \in p_j]]] \wedge$
  $\forall p_i \in P_2 : [\exists p_j \in P_1 : [p_i = p_j \longrightarrow \forall d_k \in p_i : [d_k \in p_j]]]$

There are a number of rules regarding predicates and details. Consider a system S. Let p_1 and p_2 be predicates with $p_1 = p_2$.

- $\forall d_k \in p_1 : [d_k \in p_2] \wedge p_2(S) \Leftrightarrow p_1(S)$

- $\forall d_k \in p_1 : [d_k \in p_2] \wedge \neg p_1(S) \Leftrightarrow \neg p_2(S)$

A number of assumptions need to be taken into account when regarding this formalization:

- Consider a requirement R with set of predicates P. Let p_1 and p_2 be parts with $p_1 \in P$ and $p_2 \in P$. Then, for any system S, the following statement holds:
  $p_1(S) \nRightarrow \neg p_2(S)$
  In other words, requirements cannot be internally inconsistent.

- Consider a requirement R with a set of predicates P, containing p_1 to p_n. Then, the following statement holds:
  $\exists S \in \mathcal{U} : [p_i(S)] \mid i \in 1 \ldots n$
  In other words, requirements have to be satisfiable.

5.3.2 Formalization of TracedTo

The `traced to` relation does not have well-defined semantics. The SysML specifications discourage the use of this relation in combination with other types of relations. For this approach, the use of `traced to` is discouraged altogether. It is also not formalized.

5.3.3 Formalization of DerivedFrom

The `derived from` relation denotes that one requirement is derived from a number of other requirements. Satisfying the derived requirement implies that the requirements which it was derived from are also satisfied.

Let R_1, R_2 through R_n (with $N \geq 2$) be requirements with sets of predicates P_1, P_2 through P_n respectively. R_1 is `Derived from` R_2 through R_n if and only if the following statements hold:

- $P_m \subseteq P_1 \mid m \in 2 \ldots n$
- $\forall p_i \in P_1 : [\exists m : [p_i \in P_m]] \mid m \in 2 \ldots n$
- $\forall p_j \in P_m : [\forall p_i \in P_1 : [p_i = p_j \longrightarrow \forall d_k \in p_j : [d_k \in p_i]]] \mid m \in 2 \ldots n$
- $\exists p_j \in P_m : [\exists p_i \in P_1 : [p_i = p_j \wedge \exists d_k \in p_i : [d_k \notin p_j]]] \mid m \in 2 \ldots n$

The first and second statement show that R_1 has all of the predicates of the other requirements combined, and no more. The third statement shows that all of these predicates have at least all of the details of the parts of the requirements which it is derived from. The fourth statement shows that for every requirement R_1 is derived from, there is at least one detail of a predicate in P_1 which is not in a predicate of the other requirement.

5.3.4 Formalization of ComposedBy

The `composed by` relation denotes that a requirement is contained in another requirement. The containing requirement is, in turn, completely or partially composed of all of its contained requirements.

Let R_1, R_2 through R_n (with $n \geq 2$) be requirements with sets of predicates P_1, P_2 through P_n respectively. R_2 through R_n are `composed by` R_1 if and only if statement the following statements hold:

- $P_m \subset P_1 \mid m \in 2 \ldots n$
- $\forall p_j \in P_m : [\forall p_i \in P_1 : [p_i = p_j \longrightarrow \forall d_k \in p_i : [d_k \in p_j]]] \mid m \in 2 \ldots n$
- $\forall p_j \in P_m : [\forall p_i \in P_1 : [p_i = p_j \longrightarrow \forall d_k \in p_j : [d_k \in p_i]]] \mid m \in 2 \ldots n$

The first statement shows that all predicates in the requirements contained are also in the containing requirement. The second and third statements show that the details in all corresponding predicates are the same.

5.3.5 Formalization of CopyOf

A `copy of` relation denotes that one requirement is a copy of another.

Let R_1 and R_2 be requirements with parts P_1 and P_2 respectively. R_1 is `copy of` R_2 if and only if the following statements hold:

- $P_1 = P_2$
- $\forall p_i \in P_1 : [\forall p_j \in P_2 : [p_i = p_j \longrightarrow \forall d_k \in p_i : [d_k \in p_j]]]$
- $\forall p_i \in P_1 : [\forall p_j \in P_2 : [p_i = p_j \longrightarrow \forall d_k \in p_j : [d_k \in p_i]]]$

The first statement shows that the predicates of both requirements are equal. The second and third statements show that the details of all the predicates are also equal. The `copy of` relation is reflexive, symmetric and transitive. This definition is not as strict as in SysML, where there is a difference between slave and master requirements, with slave requirements being unmodifiable. This property is of no concern for this approach, however.

5.4 Domain change case analysis

As specified in section 3.2, the following domain changes are considered in this approach:

- New requirement added
- Existing requirement removed
- Requirement made more specific
- Requirement made more abstract
- Part removed from requirement
- New part added to requirement

These are based on the formalization of the concept of requirement, as provided earlier in this chapter. The domain changes is always first applied to exactly one requirement. Each domain change is a specific change to either the parts of a requirement, or the details. Note that since applying change in this process is partly manual, mistakes can be made. Specifically, a change may be seen as a domain change where actually it is only a model change. The exact difference is this: A change to the model is only a domain change if it alters the sets of systems which satisfy the model. Formally, this can be denoted follows. Let M_b be the model before the change, and M_a the model after the change.

$\exists S \in \mathcal{U} : [(M_b(S) \wedge \neg M_a(S)) \vee (M_a(S) \wedge \neg M_b(S))]$

In the following sections, P_b indicates the parts of a requirement before the change, where P_a denotes those parts afterwards.

5.4.1 New requirement added

A domain change is classified as "new requirement added" if and only if the following statements hold:

- $P_b = \varnothing$
- $P_a \neq \varnothing$

5.4.2 Existing requirement removed

A domain change is classified as "removing an existing requirement" if and only if the following statements hold:

- $P_b \neq \varnothing$
- $P_a = \varnothing$

5.4.3 Requirement made more specific

A domain change is classified as "requirement is made more specific" if and only if the following statements hold:

Let detail $e\{< D >\}$ be added to predicate q.

- $P_b = P_a$
- $D = \varnothing$
- $\forall p_j \in P_b : [\forall p_i \in P_a : [p_i = p_j \longrightarrow \forall d_k \in p_j : [d_k \in p_i]]]$
- $\forall p_j \in P_b : [\forall p_i \in P_a : [p_i = p_j \longrightarrow \forall d_k \in p_i : [d_k \in p_j \vee d_k = e]]]$

5.4.4 Requirement made more abstract

A domain change is classified as "requirement is made more abstract" if and only if the following statements hold:

Let detail e be removed from predicate q_1.

- $P_b = P_a$
- $\forall p_j \in P_b : [\forall p_i \in P_a : [p_i = p_j \longrightarrow \forall d_k \in p_j : [d_k \in p_i \vee d_k = e \vee d_k \in e]]]$
- $\forall p_j \in P_b : [\forall p_i \in P_a : [p_i = p_j \longrightarrow \forall d_k \in p_i : [d_k \in p_j]]]$

5.4.5 Part removed from requirement

A domain change is classified as "part removed from requirement" if and only if the following statements hold:

Let predicate q be removed.

- $P_a = P_b - q$
- $\forall p_j \in P_b : [\forall p_i \in P_a : [p_i = p_j \longrightarrow \forall d_k \in p_j : [d_k \in p_i]]]$
- $\forall p_j \in P_b : [\forall p_i \in P_a : [p_i = p_j \longrightarrow \forall d_k \in p_i : [d_k \in p_j]]]$

5.4.6 New part added to requirement

A domain change is classified as "new part added to requirement" if and only if the following statements hold:

Let predicate $q\{< D >\}$ be added.

- $P_a = P_b + q$
- $D = \varnothing$
- $\forall p_j \in P_b : [\forall p_i \in P_a : [p_i = p_j \longrightarrow \forall d_k \in p_j : [d_k \in p_i]]]$
- $\forall p_j \in P_b : [\forall p_i \in P_a : [p_i = p_j \longrightarrow \forall d_k \in p_i : [d_k \in p_j]]]$

5.5 Propagation rule derivation

The derivation of the external inconsistency propagation rules as shown in chapter 4 is based on the formalization of requirement, the relation types and the domain changes. Every combination of relation type and domain change requires an external inconsistency propagation rule. This derivation is performed with these rationale.

1 The rules guarantee that the (remaining) relations in the resulting requirements model are all still valid

2 When a domain change specifies that a part or detail is removed from a requirement, it should be removed from the entire requirement model

This derivation is shown here, grouped first by domain change.

5.5.1 New part added to requirement

A domain change is classified as "new part added to requirement" if and only if the following statements hold:

Consider predicate q_1 to be added.

- $P_a = P_b + q_1$
- $\forall p_j \in P_b : [\forall p_i \in P_a : [p_i = p_j \longrightarrow \forall d_k \in p_j : [d_k \in p_i]]]$
- $\forall p_j \in P_b : [\forall p_i \in P_a : [p_i = p_j \longrightarrow \forall d_k \in p_i : [d_k \in p_j]]]$

Consider the composition relation. It is formalized as:

Let R_1, R_2 through R_n (with $n \geq 2$) be requirements with parts P_1, P_2 through P_n respectively. R_2 through R_n are `composed by` R_1 if and only if statement the following statements hold:

- $P_m \subset P_1 \mid m \in 2 \ldots n$
- $\forall p_j \in P_m : [\forall p_i \in P_1 : [p_i = p_j \longrightarrow \forall d_k \in p_i : [d_k \in p_j]]] \mid m \in 2 \ldots n$
- $\forall p_j \in P_m : [\forall p_i \in P_1 : [p_i = p_j \longrightarrow \forall d_k \in p_j : [d_k \in p_i]]] \mid m \in 2 \ldots n$

First, consider the domain change being applied to R_2. This implicates that P_2 corresponds to P_b before the change and to P_a after the change.

There are two possible scenarios. For each scenario, the two rationale are considered to determine necessary action:

Scenario 1: $q_1 \in P_1$
This implies:

- $P_2 \subseteq P_1$

In this scenario, the change made is not a domain change as it does not alter the set of systems which satisfy the model.

Scenario 2: $q_1 \notin P_1$
This implies:

- $P_2 \not\subset P_1 \wedge P_2 \neq P_1$

In this scenario, the relation no longer holds. Here, we propagate the domain change to R_1.

Consider predicate q_2 to be added.

- $P_a = P_b + q_2$
- $\forall p_j \in P_b : [\forall p_i \in P_a : [p_i = p_j \longrightarrow \forall d_k \in p_j : [d_k \in p_i]]]$
- $\forall p_j \in P_b : [\forall p_i \in P_a : [p_i = p_j \longrightarrow \forall d_k \in p_i : [d_k \in p_j]]]$

Then, for R_1 we have P_{1b}, the predicates before the new change has been applied, and P_{1a}, the predicates after the new change has been applied.

Then we have, $P_{1a} = P_{1b} + q_2$. If the change is correctly implemented, $q_1 = q_2$. I.e. the addition to the R_1 is the same as the addition to R_2. If so, the statements which have to hold for the composition relation between R_1 and R_2 to be valid, hold.

From this analysis, we can determine that there are two possible propagation scenarios:

- No propagation
- Apply domain change "New part added to requirement" to R_2

Next, consider the domain change being applied to R_1. This implicates that P_1 corresponds to P_b before the change and to P_a after the change. Here, the statements which much hold for the composition relation between R_1 and R_2 through R_n to be valid, hold.

From this analysis, we can determine that there is only one possible propagation scenario:

- No propagation

5.5.2 Requirement made more abstract

A domain change is classified as "requirement made more abstract" if and only if the following statements hold:

Let detail e be removed from predicate q_1.

- $P_b = P_a$
- $\forall p_j \in P_b : [\forall p_i \in P_a : [p_i = p_j \longrightarrow \forall d_k \in p_j : [d_k \in p_i \vee d_k = e \vee d_k \in e]]]$
- $\forall p_j \in P_b : [\forall p_i \in P_a : [p_i = p_j \longrightarrow \forall d_k \in p_i : [d_k \in p_j]]]$

Consider the derivement relation. It is formalized as:

Let R_1, R_2 through R_n (with $n \geq 2$) be requirements with sets of predicates P_1, P_2 through P_n respectively. R_1 is `Derived from` R_2 through R_n if and only if the following statements hold:

- $P_m \subset P_1 \mid m \in 2 \ldots n$

- $\forall p_i \in P_1 : [\exists m : [p_i \in P_m]] \mid m \in 2 \ldots n$
- $\forall p_j \in P_m : [\forall p_i \in P_1 : [p_i = p_j \longrightarrow \forall d_k \in p_j : [d_k \in p_i]]] \mid m \in 2 \ldots n$
- $\exists p_j \in P_m : [\exists p_i \in P_1 : [p_i = p_j \wedge \exists d_k \in p_i : [d_k \notin p_j]]] \mid m \in 2 \ldots n$

First, consider the domain change being applied to R_1. This implicates that P_1 corresponds to P_b before the change and to P_a after the change. Then, for each requirement R_2 through R_n, the possible scenarios are the same, so we only consider R_2 here.

Scenario 1: $q_1 \notin P_2$
In this scenario, the statements which have to hold for the relation between R_1 and R_2 to be valid, hold.

Scenario 2: $q_1 \in P_2 \wedge p \in P_2 : [p = q_1 \wedge e \in p]$
In this scenario, the statements which have to hold for the relation between R_1 and R_2 to be valid, hold. However, considering the first propagation rule derivation rationale, since the detail is removed from R_1 and is present in R_2, it should also be removed there.

Scenario 3: $q_1 \in P_2 \wedge p \in P_2 : [p = q_1 \wedge e \notin p]$
In this scenario, the fourth statement which have to hold for the relation between R_1 and R_2 to be valid, may or may not still hold. This can be evaluated by the following statement:

- $\exists p_j \in P_2 : [\exists p_i \in P_1 : [p_i = p_j \wedge \exists d_k \in p_i : [d_k \notin p_j \wedge d_k \neq e]]]$

In natural language, R_1 needs to have another detail apart from e which makes it a derivation of R_2. If the previous statement evaluates to false, it does not and the relation is no longer valid. Otherwise, there are no inconsistencies.

In both cases however, there is no external inconsistency propagation.

From this analysis, we can determine that there are two possible propagation scenarios:

- Apply domain change "requirement made more abstract" to R_2
- No propagation

Note that in case the change is not propagated, the validity of the relation needs to be checked.

5.6 Discussion of formalization

This section describes some of the alternatives we considered. The crucial difference between all possible formalization approaches lies in the formalization of the requirements.

5.6.1 Using predicates and systems

[GKvdB09] formalizes requirements using predicates, similar to us, and systems. This is most likely is sufficient. Formalizing the relations is possible and adequate, as well as deriving the external inconsistency propagations rules. In fact, the difference between using predicates and systems and the current approach lies only in the explicit nature of the details. However, we believe that using parts and details as opposed to predicates and systems makes the derivation of the external inconsistency propagation rules more clear. As stated before, domain changes include removing and adding parts and details. Directly incorporating these in the formalization of the requirements and the relations creates a much clearer link between these formalizations and the derivation of the rules. Another thing to note is that there is some redundancy when using predicates and systems, as the systems which satisfy a requirement can be derived from its predicates.

5.6.2 Using systems only

We also attempted to create a formalization of each of the relevant model elements using only systems. This created a problem concerning the `derived from` and the `composed by` relation types. Without further information, the `derived from` relation type only states the necessity of a subset relation between the systems of two requirements. This would however, by definition create a `derived from` relation to a composed requirement from each of its components.

Since the SysML specifications are not entirely clear on the definitions of these relation types, this could be interpreted as perfectly fine formalizations. However, we felt it would increase the value of the entire metamodel if the `derived from` were not just a reversed `composed by` relation. This is why we needed additional information.

5.7 Summary

This chapter first formalized requirements as a collection of parts which can have details applied to them. Using this formalization, the various relation types were formalized; `DerivedFrom`, `ComposedBy` and `CopyOf`. The domain changes were formalized as the difference between the parts and details of a single requirement before and after the domain change has been determined.

Next, rules with which the external inconsistencies are propagated were determined. Only a small number of scenarios were described. These scenarios consist of one domain change, two requirements and one relation between the two requirements. The rule determination was done by combining the formalizations of requirement, relations and domain changes. We also shortly discussed alternative requirement formalizations.

The mathematical determination of the propagation rules does not exhaust all scenarios. Also, the determination of the currently handled scenarios is incomplete. The text states that the statements required for the relation to hold, still or no longer hold. However, the mathematical proof for these remarks is omitted. Finally, the current description of predicates and details allows for modeling situations in which multiple choices are feasable.

6 Example process usage

In this chapter we show how the process described in the previous chapters can be used in practice.

6.1 Example Model

As an example model we use a subset of a fictional Course Management System for a fictional university [CMS]. There are four stakeholders in this system:

- Students
- Lecturers (in the example model shown here labeled as teachers)
- Administrators
- System maintainers

The main subject of the system are the courses. Course information such as roster, lecturer, enrolled students and news messages can be managed, as well as personal information regarding students and lecturers. Note that the example intentionally contains some conflicting requirements, mostly between different stakeholders. For each stakeholder, there are also a number of non-functional requirements, such as privacy, security and interoperability.

We show the working of our process using a subsection of the requirements which regards the functionality for the lecturers to manage dynamic course information. This involves the following:

- Posting news messages for a course. Only students enrolled in that course can read them.
- Managing the course archive. The archive contains items such as homework assignments and lecture sheets.

- Managing student teams. Lecturers can create student teams for a course and manage them.
- Managing student grades for a course.
- Importing course information from one year into another, so that the lecturer doesn't have to re-enter a lot of the information again every year.

We chose this subset because it is relatively easy to understand and because it contains both `DerivedFrom` and `ComposedBy` relations. The example does not contain the `CopyOf` relation, no conflicting requirements and no non-functional requirements.

Figure 6.1 shows the requirements model of the subsection of the Course Managament System in a graphical notation. The specifics can be found in appendix B.

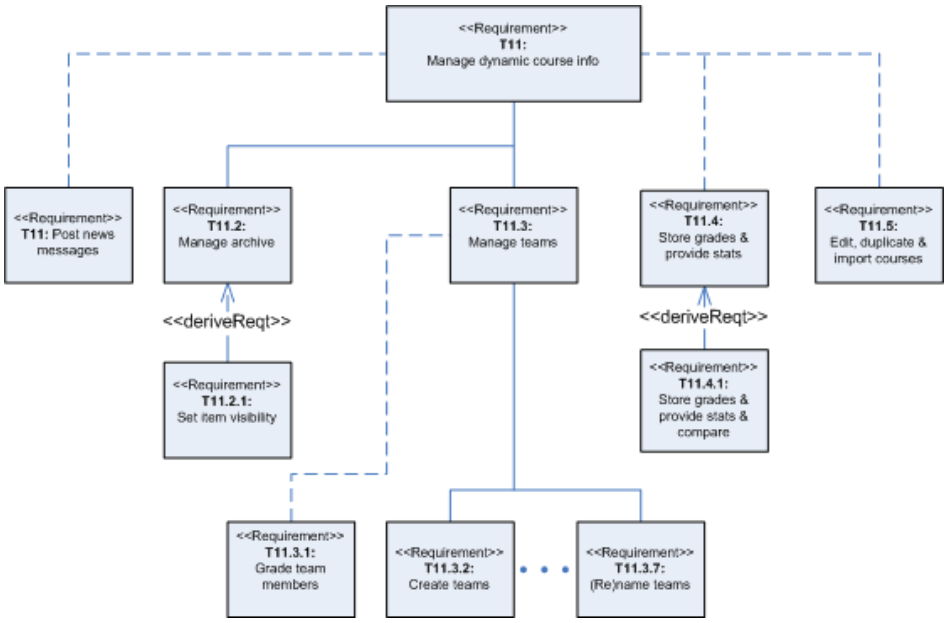

Figure 6.1: Example Requirements model

The dots in the figure between T11.3.2 and T11.3.7 mark the requirements T11.3.3 to T11.3.6. The dotted lines in the figure indicate that there is some relation between the predicates of the requirements connected with this line, but this relation is not of a type specified in SysML.

6.2 Example changes

As specified in chapter 4, the change management process starts with a domain change being identified by a requirements engineer. This section contains three example domain changes which will be managed by the process:

- Teachers will no longer be allowed to set the visibility of items in the archive
- Student teams need no longer be supported by the system
- Teachers have to be able to remove grades as well as storing them

6.2.1 Remove setting visibility of archived items

As specified by the process, the domain change first has to be decomposed into an atomic domain change, but how this is done is not elaborated upon. This specific domain change however, can only be decomposed and applied correctly in one way: By applying the domain change "Requirement made more abstract" to requirement T11.2.1.

The domain change is then mapped automatically to the external inconsistency "Requirement in the model more specific than in the domain", as stated in table 4.1, and applied to T11.2.1. The process then picks requirement T11.2 to be handled next, as it is the only related requirement. The scenario in this case is: T11.2.1 is designated externally inconsistent as "Requirement in the model more specific than in the domain" and is related to T11.2 by the `DerivedFrom` relation, with T11.2.1 as the source. As specified in table 4.2, there are two propagation possibilities:

- To apply the external inconsistency "Requirement in the model more specific than in the domain" to T11.2
- Not to propagate an external inconsistency

This choice is made by the requirements engineer, who we assume makes the second choice, being the correct one for this domain change. Note that table 4.2 contains a foot-note on this scenario, namely that the two requirements may have become identical. It is up to the requirements engineer to solve this issue.

Since no other requirements are related to T11.2.1 and T11.2 is designated not externally inconsistent, the propagation ends. The process then maps the inconsistency to a proposed model change, according to table 4.1. The result is shown in table 6.2.1.

Requirement	**External Inconsistency**	**Proposed model change**
T11.2.1	Requirement in the model more specific than in the domain	Detail removed from requirement

Table 6.1: Process result for example one

The proposed model change is shown to the requirements engineer, who will then choose if and how to implement it. No further change management support is provided.

6.2.2 Remove student team support

Again, the requirements engineer has to decompose and apply the domain change. In this case, the fact that the system needs no longer support student teams implies that teachers need no longer be able to manage them. This domain change can be handled in several ways:

- Applied to requirement T11.3 as "Existing requirement removed"
- Applied to requirement T11 as "Part removed from requirement"
- Applied to requirements T11.3.2 to T11.3.7 as "Existing requirement removed"

It is up to the requirements engineer to choose which is most practical, but note that requirement T11.3.1 needs also be designated as "Existing requirement removed", as it is not related to any other requirement in the model by a SysML specified relation. Whichever approach the engineer chooses, the results are the same, provided the correct propagation possibilities are selected. For this example, we assume the domain change "Existing requirement removed" is applied to T11.3. The other options are not elaborated upon here.

The domain change is automatically mapped to the external inconsistency "Requirement not in the domain, but present in the model", as stated in table 4.1, and applied to T11.3. The process then randomly picks a requirement related T11.3 to handle next, but whichever order the requirements are handled in, the results are the same given that the requirements engineer selects the correct propagation possibilities. We assume T11.3.2 is picked first.

The scenario being handled in this particular situation is: T11.3 is designated externally inconsistent as "Requirement not in the domain, but present in the model" and is related to T11.3.2 by the `ComposedBy` relation, with T11.3 as the source. As specified in table 4.3, there is only one propagation possibility: To apply the same external inconsistency to T11.3.2.

After this step, the process selects another random requirement related to either T11.3 or T11.3.2. As T11.3.2 is not related to any requirement which has not been processed yet, a requirement related to T11.3 will be selected. We assume the process first selects the requirements T11.3.3 to T11.3.7. These requirements are handled in exactly the same manner as T11.3.2.

Finally, the process will select T11. The scenario in this case is: T11.3 is designed externally inconsistent as "Requirement not in the domain, but present in the model" and is related to T11 by the `ComposedBy` relation, with T11 as the source. As specified in table 4.3, there is only one propagation possibility: To apply the external inconsistency "Requirement in the model has more parts than in the domain" to T11.

As a last step, the process selects T11.2. The scenario in this case is: T11 is designated externally inconsistent as "Requirement in the model has more parts than in the domain" and is related to T11.2 by the `ComposedBy` relation, with T11 as the source. As specified in table 4.3, there are three propagation possibilities:

- To apply the external inconsistency "Requirement not in the domain, but present in the model" to T11.2

- To apply the external inconsistency "Requirement in the model has more parts than in the domain" to T11.2
- Not to propagate an external inconsistency

This choice is made by the requirements engineer, who we assume makes the third choice, being the right one for this domain change. At this point, as there are no requirements related to requirements with an external inconsistency applied to them which have not yet been handled, the propagation ends.

The next step is for the process to automatically map the external inconsistencies to proposed model changes. This mapping is done according to table 4.1 and the results are shown in the table 6.2.2.

Requirement	**External Inconsistency**	**Proposed model change**
T11	Requirement in the model has more parts than in the domain	Part removed from requirement
T11.3	Requirement not in the domain, but present in the model	Requirement is removed
T11.3.2	Requirement not in the domain, but present in the model	Requirement is removed
T11.3.3	Requirement not in the domain, but present in the model	Requirement is removed
T11.3.4	Requirement not in the domain, but present in the model	Requirement is removed
T11.3.5	Requirement not in the domain, but present in the model	Requirement is removed
T11.3.6	Requirement not in the domain, but present in the model	Requirement is removed
T11.3.7	Requirement not in the domain, but present in the model	Requirement is removed

Table 6.2: Process result for example three

The proposed model changes are shown to the requirements engineer, who will then choose if and how to implement them. No further change management support is provided.

6.2.3 Add support for removing grades

Here, we consider the functionality for teachers to remove grades to be part of managing dynamic course information, as stated in T11. As such, the domain

change "Part added to requirement" is applied to this requirement. After the domain change is mapped to the external inconsistency "Requirement in the model has less parts than in the domain", the propagation starts in the same manner as in the above two examples. Both T11.2 and T11.3 will be considered, but the propagation rules state that there is no propagation to these two requirements. The final result would be a model change to T11, adding the new functionality.

Although the end result would lead to a model which is both externally and internally consistent (the model correctly reflects the domain and is not in conflict with itself) the requirements engineer may decide it would improve the model if the part was also added to some sub-requirement of T11. This could be T11.4 or a new requirement. Here, we assume the requirements engineer decides to apply the same domain change applied to T11 to T11.4 as well.

The steps of the process are similar to those in the previous examples. The domain change is mapped to the external inconsistency "Requirement in the model has less parts than in the domain". This external inconsistency is propagated to T11.4.1 and then mapped to proposed model changes. The results of both domain change applications combined are shown in table 6.2.3.

Requirement	**External Inconsistency**	**Proposed model change**
T11	Requirement in the model has less parts than in the domain	Part added to requirement
T11.4	Requirement in the model has less parts than in the domain	Part added to requirement
T11.4.1	Requirement in the model has less parts than in the domain	Part added to requirement

Table 6.3: Process result for example three

The proposed model changes are shown to the requirements engineer, who will then choose if and how to implement them. No further change management support is provided.

6.3 Summary

This chapter showed how the process described earlier in this thesis can be used in practice. We started by showing a simple example model regarding a course management system. Next, we applied in small steps three different feasable domain changes which have to be reflected in the model. The steps taken show how the domain change is applied, how the external inconsistency propagation works and how these are then mapped to proposed model changes.

One issue which should be noted is that there are a number of relations between requirements in the model which do not conform to any relation type

in SysML. This reduces the amount of information which serves as input to the process, but it is a limitation of SysML, not of our process.

7 Tool Support

This chapter describes the tool Blueprint and how our solution is implemented as an extension to this tool.

7.1 Blueprint

Blueprint is a software modeling tool built by @-portunity [@-p] as an Eclipse application, which supports Model Driven Architectures and fully complies with OMG's MetaObject Facility, also known as MOF. It aims to be a complete software modeling tool. That is, it is intended to be the only tool necessary to model at all levels of the software project. By adhering to OMG standards, it allows users to easily develop and exchange information.

More information on the tool can be found here [Blu].

7.1.1 Blueprint capabilities

MOF compliance implies that Blueprint supports modeling using numerous OMG standards, most notably UML, SyML and OCL. The three mentioned standards are inherently supported, which means that Blueprint can be used to create class diagrams, sequence diagrams and requirements diagrams. However, other MOF-based standards can be imported and used for modeling as well. By doing so, Blueprint allows developers to create models on any abstraction level, ranging from requirements to designs to code. These models can then be related, which helps parse, transfer, store and transform them.

Blueprint also allows easy creation and application of UML and SysML profiles. As explained in section 2.2, profiles allow developers to extend and adapt UML and SysML to their specific needs.

Customizability is also a focal point of Blueprint, allowing developers to not only apply their own MOF-based models and metamodels, but also allowing them to create their own modeling tools for these models. This can be very

useful in making development less complex. Since it is built as an Eclipse application, plugins can also be created to further customize the tool.

Blueprint also supports importing and exporting models to several formats, including Java Archives, XMI and EMF, allowing Blueprint to cooperate with other tools already in use in the industry.

7.1.2 Blueprint data structure

At the basis of the datastructure in Blueprint is the project. Projects contain any number of phases, which in turn contain subphases which contain models and metamodels. Projects, phases and subphases are named and can be documented with dedicated documentation functionality. Figure 7.1 shows the data structure graphically.

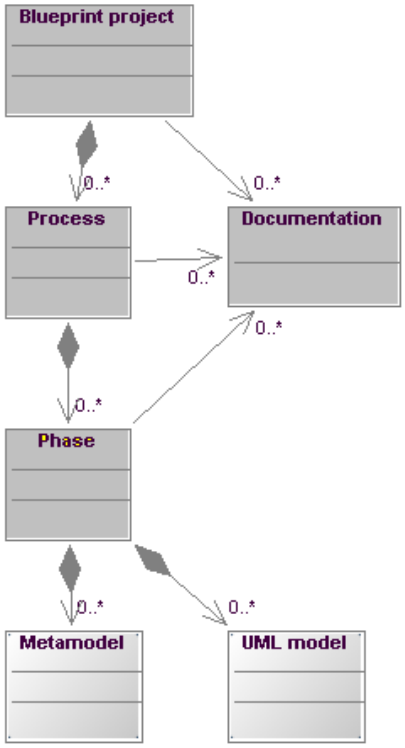

Figure 7.1: Blueprint project structure

The rest of the datastructure depends on the metamodel(s) used.

7.2 Solution location

We implemented our solution as a plugin to Blueprint. The plugin was created as an extension to the user interface, allowing the user to select how to initialize the change impact analysis process.

Currently, the input required to start the process is given through the graphical user interface. The other input and output is performed using the standard in and standard out respectively. Obviously, this could be improved, but it is sufficient for a prototype.

Figure 7.2 shows an example of how the plugin could be used. On the right it shows the requirements model. The menu the user has selected allows him to apply a domain change to the selected requirement in order to start the process.

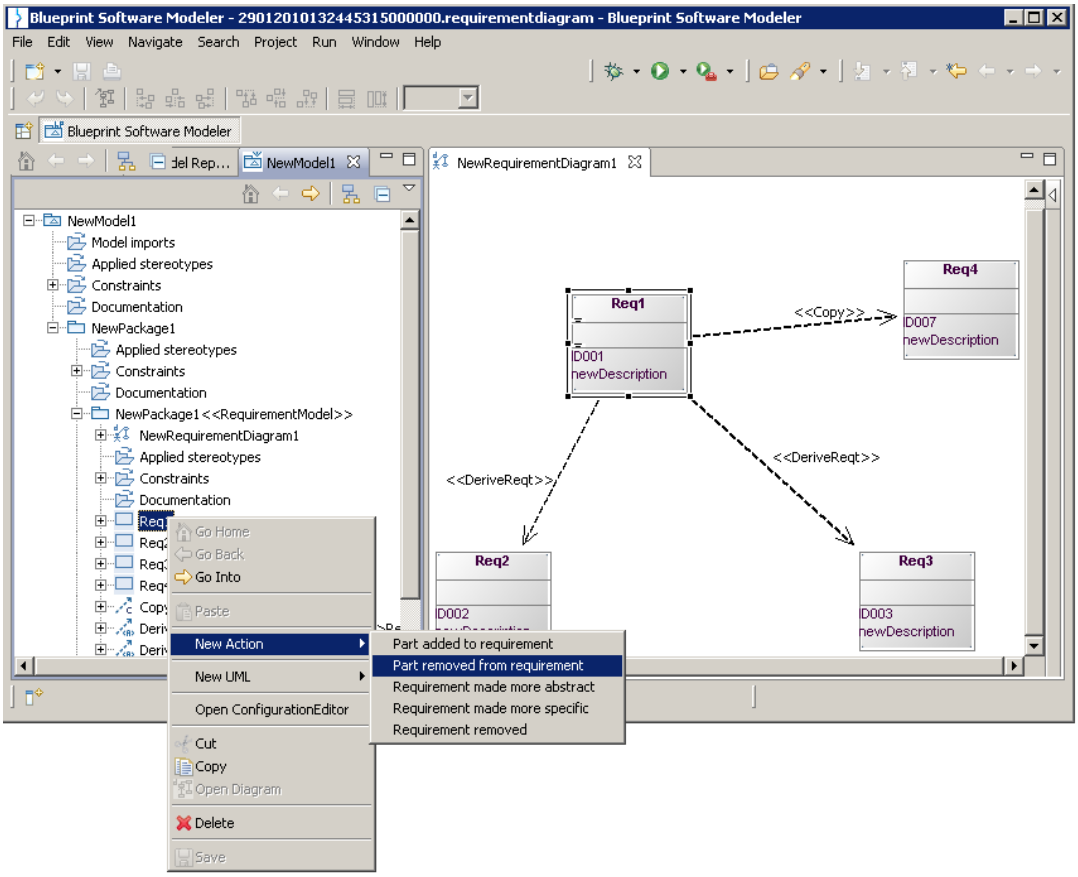

Figure 7.2: User Interface example

7.3 Plugin development

The development of the plugin was performed in small phases. A set of requirements was provided by us showing what was necessary for the next step in the implementation of the solution. @-portunity satisfied these requirements, which allowed us to continue our work and come up with a new set of requirements, iteratively.

The test plan for the plugin can be found in appendix A.

7.4 Plugin architecture

Before we describe the plugin architecture, a short note on the standard Eclipse plugin architecture. A plugin requires a plugin.xml file. This xml-file denotes an extension point, which contains all actions for the plugin. These actions contain descriptions of where the action is located within the menu structure, a name, a label and a class associated with it. Of this class, which have to extend the **AbstractUmiModelCommandAction**-class, the `run()` method is called, which initiates the action.

Figure 7.3 shows the current architecture of the plugin, which we explain below.

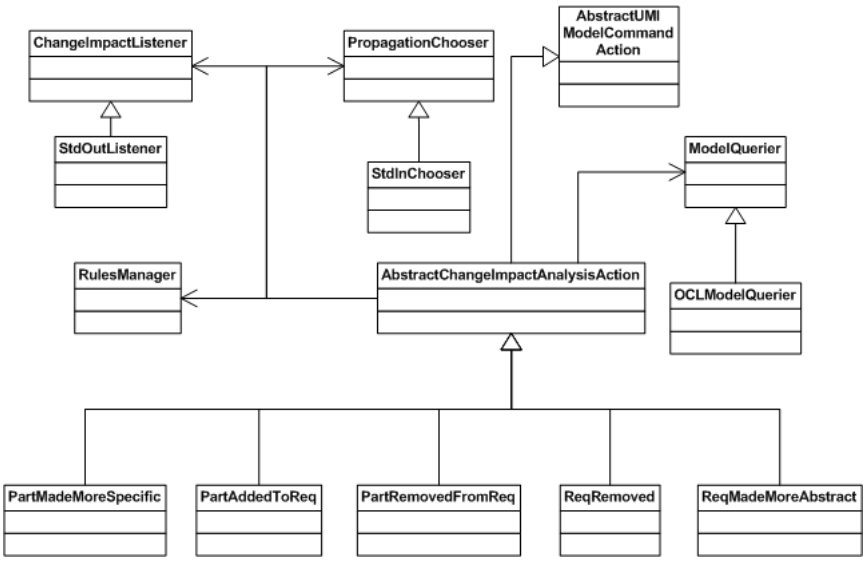

Figure 7.3: Plugin architecture

7.4.1 AbstractChangeImpactAnalysisAction

This class directs the steps of the entire process. It starts by determining the domain change and, using that, the external inconsistency applied to the selected requirement. This external inconsistency is then propagated throughout the model iteratively. It uses the **ModelQuerier**-interface to query the model for elements related to other elements. It uses the **RulesManager**-class to determine the possible choices of propagation. It uses the **PropagationChooser**-interface to choose between propagations and the **ChangeImpactListener**-interface to update listeners of the entire process's progress and results, both intermediate and final.

7.4.2 PartMadeMoreSpecific

This is a subclass of the **AbstractChangeImpactAnalysisAction**-class and is invoked by the Eclipse plugin framework. Its only purpose is to allow the

superclass to determine the starting domain change.

7.4.3 PartMadeMoreAbstract

This is a subclass of the **AbstractChangeImpactAnalysisAction**-class and is invoked by the Eclipse plugin framework. Its only purpose is to allow the superclass to determine the starting domain change.

7.4.4 PartAddedToReq

This is a subclass of the **AbstractChangeImpactAnalysisAction**-class and is invoked by the Eclipse plugin framework. Its only purpose is to allow the superclass to determine the starting domain change.

7.4.5 PartRemovedFromReq

This is a subclass of the **AbstractChangeImpactAnalysisAction**-class and is invoked by the Eclipse plugin framework. Its only purpose is to allow the superclass to determine the starting domain change.

7.4.6 ReqRemoved

This is a subclass of the **AbstractChangeImpactAnalysisAction**-class and is invoked by the Eclipse plugin framework. Its only purpose is to allow the superclass to determine the starting domain change.

7.4.7 ModelQuerier

The **AbstractChangeImpactAnalysisAction**-class needs to query the model in order to propagate external inconsistencies. Querying which model elements are affected is done using the **ModelQuerier**-interface. This interface specifies a single method which finds all elements that are related to a single element by a specific type of relation. Currently, only the **OCLModelQuerier**-class implements this interface.

7.4.8 OCLModelQuerier

This class provides functionality to query the model using OCL. It uses predefined, parameterized OCL queries.

7.4.9 RulesManager

The **RulesManager** determines possible external inconsistency propagation scenarios depending on four parts of information:

- An external inconsistency
- A relation type
- Whether it is the source or the target of the relation that is externally inconsistent

- The multiplicity of the relation

The rules themselves correspond to those described in section 4.3.

7.4.10 PropagationChooser

The **AbstractChangeImpactAnalysisAction**-class needs to make choices regarding external inconsistency propagation. The **PropagationChooser**-interface specifies a function which requires a source element, a target element, an external inconsistency and a number of possible choices that can be used to make a choice. Currently, only the **StdInChooser**-class implements this interface.

7.4.11 StdInChooser

This class relays possible external inconsistency choices to the user using the standard input and output of the process.

7.4.12 ChangeImpactListener

The **ChangeImpactListener**-interface is used by the **AbstractChangeImpactAnalysisAction**-class to notify listeners of the process's progress and results. There are five distinct events:

- Process started. Information provided consists of the starting domain change and the starting model element
- Domain change mapped to external inconsistency. Information provided consists of the domain change, the external inconsistency and the model element
- External inconsistency propagated. Information provided consists of the source and target model elements and their respective external inconsistencies
- External inconsistencies mapped to model changes. Information provided consists of all model elements to which updates are proposed and the specific type of update for each one of them
- Requirement marked for evaluation. Information provided consists of the model element which will be evaluated in the future

Currently, only the **StdOutListener**-class implements this interface.

7.4.13 StdOutListener

This class notifies the user of the progress and results of the process by using the process's standard output.

7.4.14 AbstractUMIModelCommandAction

This is the abstract Eclipse plugin framework class which all plugins need to extend.

7.5 Example usage

This section describes an example usage of the current prototype, using an example requirements model. For each step, the related architectural elements are noted.

7.5.1 Example model

Figure 7.4 shows the example model used during this example usage of the tool.

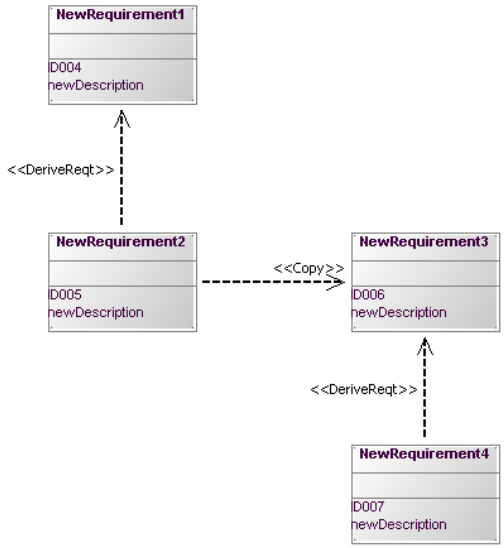

Figure 7.4: Example model

7.5.2 Usage

This section describes the usage of the tool. For information on how the process described in this thesis is used, see chapter 6.

The process is started by the user entering the primitive domain change. For the tool, this is done by specifying one requirement which the domain change has made externally inconsistent, and how. In this case, we specify that the external inconsistency "Requirement made more abstract" is applied to R_1. Entering the initial external inconsistency is done by creating a new object of one of the subclasses of the **AbstractChangeImpactAnalysisAction**. In this case, the **PartMadeMoreAbstract**.

Firstly, the **ChangeImpactListener** is used to inform listeners the process has started and that R_2 is externally inconsistent as specified by the "Requirement made more specific"-inconsistency. Next, the **ModelQuerier** is used to find the model elements directly related to R_2, in this case R_1 and R_3. The process randomly picks which requirement to evaluate first, here we assume it first

picks R_1. The **RulesManager** is used to determine the possible propagations from R_2 to R_1. These are:

- No propagation
- The external inconsistency is propagated to R_1

The user will have to make this choice, and for this purpose the **PropagationChooser** is used. The **StdInChooser** implements this interface and will propose the two choices to the user, along with the scenario information. Here we assume the user selects the first option.

Next, the process will evaluate requirement R_3. The **RulesManager** is once again used to determine the possible propagation choices. In this case, there is only one; to propagate the external inconsistency from R_2 to R_3. Because there is only one possibility, no **PropagationChooser** is not used.

After these steps, the **ModelQuerier** is once again used, this time to find the elements related to R_3. These are R_2 and R_4, but R_2 has already been evaluated, so it will be ignored for the rest of the external inconsistency propagations. As for R_4, the **RulesManager** specifies only one possibility; to propagate the external inconsistency from R_3 to R_4.

As the last step in the external inconsistency propagation, R_4 is evaluated the same way as previously described. As R_4 is not related to any unevaluated requirements, there is no more propagation.

Finally, the process maps all external inconsistencies to proposed model changes using the **RulesManager**. This is where the process ends, choosing whether to implement the changes and actually doing so is done entirely by the user.

All the while during the process, the **ChangeImpactListener** is used to update the user of the process progress. The proposed model changes are also offered to the user using this interface.

7.6 Summary

This chapter first described the tool Blueprint as developed by @-portunity. Blueprint is a software modeling tool built as an Eclipse application, which supports Model Driven Architectures and fully complies with OMG's MetaObject Facility. Relevant to our work, it fully supports SysML modeling and since it is created as an Eclipse application, we could create our prototype as a plugin.

The rest of the chapter described the prototype, starting with the architecture. The user interface of the prototype is mostly text-based. This is not very user friendly, but it is sufficient. The architecture of the tool is set up so that it should be easy to transition from this user interface to another.

Finally, we showed how the tool is used in practice. This is in line with chapter 6, which showed the usage of the process.

8 Conclusion

Change management has always been a vital process within software development projects. Traditionally, it is performed more or less ad hoc. First, a change is requested, for example because a stakeholder presents a change in his or her interests. A software engineer uses experience and knowledge of the software to then identify which software artifacts are affected by the change, how they are affected, and how they should be updated.

Several methods have been developed over the years to support change management, but they focus mainly on managing change within code, within designs or between code and design. Other methods describe how change management can be structured as a bussiness process. We intended to support change management, or change impact analysis to be more precise, within software projects with regard to the requirements level. The requirements in the requirements level are derived from the domain, and any change in the domain can necessitate a change in the requirements.

Chapter 2 showed the basic concepts we used to answer our research questions. We used a subset of SysML, a general-purpose modeling language for systems engineering, as a modeling language for requirements. SysML is derived from the Unified Modeling Language, or UML. The elements from SysML we used are the requirement and three requirement relations, namely `CopyOf`, `DerivedFrom` and `ComposedBy`. Other relevant technologies are the MetaObject Facility, or MOF, and the Object Constraint Language, or OCL. Mof serves as the metamodel for UML and SysML, OCL is used for querying models.

8.1 Classification of changes

Our first two research questions regarded classification:

1.1 What are the types of domain changes

1.2 What are the types of impact of domain changes on model elements

We needed to answer these questions to create a basis for our work. In chapter 3, we first classified domain changes. These cause inconsistencies between the model and the domain. These external inconsistencies, as we call them, are also classified in this chapter, along with model changes to resolve them. The domain changes are classified as follows:

1 New requirement added

2 Existing requirement removed

3 Requirement made more specific

4 Requirement made more abstract

5 Part removed from requirement

6 New part added to requirement

The external inconsistency classification is parallel to this classification. For example, the impact of the domain change "Part removed from requirement" is that a requirement in the model has more parts than in the domain. The model changes are classified as follows:

1 Requirement is added

2 Requirement is removed

3 Detail added to requirement description

4 Detail removed from requirement description

5 Part removed from requirement description

6 Part added to requirement description

7 Relation is removed

8 Relation is added

Note that the bottom two are not used in our process.

8.2 Determining and resolving impact

Our next research questions focussed on determining and resolving impact of domain changes on requirements models:

2.1 How can we determine which model elements are impacted

2.2 How can we determine the type of impact on these elements

2.3 How can we solve the impact of domain changes on model elements

We determine the impact of a domain change using a dedicated semi-automatic process. We based the exact workings of the process on the formalization of relevant SysML model elements and on the formalization of domain changes. See also chapters 4 and 5.

8.2.1 Process workings

The process is partially automated and is shown in figure 4.1. It starts with a requirements engineer determining a change in the domain and splitting that domain change into domain changes which fit in the above mentioned classification. The requirements engineer then selects a model element which is impacted by the domain change. The domain change is automatically mapped to the corresponding external inconsistency and from this point, the process guides the engineer in propagating this external inconsistency through the model.

This propagation is performed according to dedicated propagation rules. These specify a number of propagation possibilities for when a requirement with an external inconsistency is related to another requirement. These propagation possibilities depend on relation type, relation direction and the type of the external inconsistency. The process requires the user to select the appropriate propagation, explained in the following subsection. After the propagation of inconsistencies has finished, the process automatically proposes model changes which could be implemented to put the model back in sync with the domain. Whether or not to implement them and actually doing so is up to the requirements engineer.

8.2.2 Formalization and derivation

The external inconsistency propagation rules are the core of the entire process. In order to derive these rules, we first needed to formalize the concept of requirement, the relation types and the domain changes. The requirement is formalized as a set of parts or predicates, which cannot be empty, with any number of details applied to them. The details can contain subdetails as well.

The difference between a part of a requirement and a detail is that the latter does not have meaning on its own. A requirement stating "The system shall perform function X in Y seconds" can be considered to have one part, namely "The system shall perform function X", with one detail "in Y seconds".

The relations are formalized as statements regarding the parts and details of two requirements. For example, the `DerivedFrom` relation is formalized along these lines (considering requirement A is derived from requirement B):

- All parts present in requirement B have to be present in requirement A
- All details present in requirement B have to be present in requirement A
- At least one detail has to be present in requirement A and absent in requirement B

The domain changes are formalized as statements regarding the parts and details of one requirement before and after a domain change. For example, the domain change "Requirement made more abstract" is formalized along these lines:

- The parts before and after the change have to be the same
- All details present after the change have to be present before the change
- One detail, which may have other details applied to it, has to be present before, but absent after the change

Combining these formalizations, we (partially) derived the external inconsistency propagation rules.

8.3 Tool support

Our final research question regards implementing our approach:

3.1 How can we use tools to support change management

We implemented our tool support as an extension of the tool BluePrint Software Modeler, created by @Portunity. BluePrint is a software modeling tool with extensive customization support, which allows users to model in any MOF-based language, including SysML.

In order to use the tool, the user first models his requirements, with SysML as the metamodel. Next, a domain change can be selected and applied to a specific requirement in the model. As stated in the change impact analysis process, this domain change is mapped to an external inconsistency, which is the starting point for the inconsistency propagation. During this propagation, the user is required to choose between different propagation possibilities. Any situations in which there is only one possibility are handled automatically. After the propagation, a number of model changes are proposed to the user.

Currently, the tool allows users to go through the entire process. Only some exceptional situations are not covered, such as when two requirements possibly become copies. Additionally, the user interface of the tool is rather primitive and not very user-friendly. The models are represented graphically, but most of the input and output is text-based. However the architecture of the tool is set up so that it should take relatively little effort to transition from this user interface to another, preferably graphical, one. Extending the tool with additional domain changes, model changes or relation types should also be relatively simple.

8.4 Evaluation

8.4.1 Choice of metamodel

We decided on using SysML as our metamodel. This has several benefits. For one, it is a standard developed by OMG, which warrants a certain amount of support. Additionally, since SysML is already in use by the industry, applying our approach is easier than when we would have used a proprietary metamodel.

There are also some downsides, however. The most important one is that the specifications of the requirements relations in SysML are very ambiguous. Hence, different users will interpret them in different ways. This means they will also expect change impact analysis to work in different ways on the relations. With our formalization, we focussed on one possible interpretation, but this may make it more difficult for users to use our approach if their interpretation of the relations is different.

8.4.2 Formalization

The formalization described in this thesis is not the first version. We started out with the same approach as [GKvdB09], but found the systems and predicates

used there to be somewhat redundant. The systems which satisfy a requirement can be derived from the predicates in the requirement. However, after some time we realized having only systems or only predicates is not sufficiently expressive for this approach.

The approach described here, with requirements consisting of parts and details, is based mostly on changes to the model and domain. Several mathematical formalizations may be applicable here, but of all those considered, the current one seemed most intuitive. When a requirements engineer makes changes to a requirements model based on domain changes, he mostly adds or removes parts, makes requirements less or more specific, or removes requirements entirely. Our domain changes and impact analysis fits this perfectly.

8.4.3 Change Impact Analysis Process

The change impact analysis process we described in this thesis is not the first version. We started with the process as shown in figure 8.1.

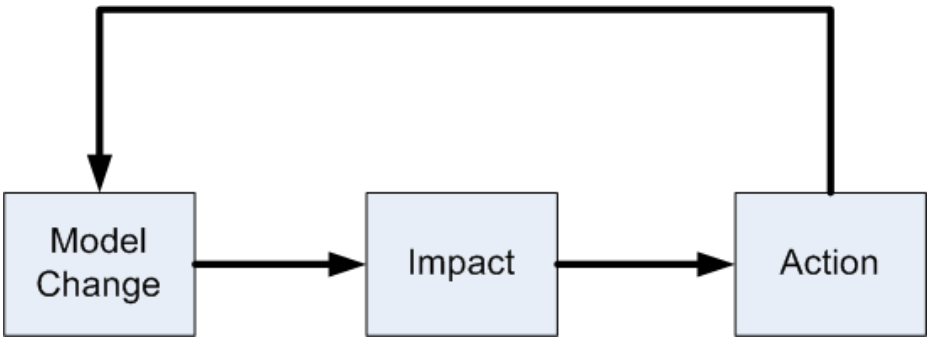

Figure 8.1: First change impact analysis process

Here, any change made to the model causes impact of some sort. This impact necessitates some action which, when performed, results in a model change. This model change again may cause impact and so on. The most important difference between this process and the one we used is that here, there is no distinction between changes made to the model for things like readability and understandability, or for a domain change. Without knowing the cause of the change, the impact calculation and propagation becomes far less precise.

Another benefit of our latest approach compared to our first, is that the actual changes made to the model to solve impact have no impact themselves. Only when all impact is calculated does the process enter the phase of actually resolving the impact. Seperating these steps makes the process easier to use.

8.4.4 Final remarks and future work

The method we propose has some limitations. First of all, it is partially automated. Some experience and knowledge of the system is still necessary and there is always a chance that this leads to errors. In the future, the process could become more automated.

Also, as stated before, the method works with a very specific interpretation of the SysML requirement relations. Rather than attempting to find an interpretation that is universily agreed on, the process could be made customizable. This would allow users to define their own relations, but using the same requirement and domain change formalizations. With these relation formalizations, new external inconsistency rules could then be calculated and used.

Another limitation is that the process does not allow composite domain changes to be applied. It is not possible to, for example, handle multiple requirements being removed from the domain at once. The composite domain change has to be split up by the engineer, who then has to apply the changes one by one. Functionality for processing composed domain changes could also be added in the future.

Apart from these limitations however, the method certainly has merit. Since we use relational semantics, we can automatically eliminate a lot of propagation scenarios. Because of this, the impact explosion is very limited compared to existing methods.

In addition to more precisely determining which requirements are impacted, the method also determines the type of impact to the model elements. Doing so provides the possibility to propose model changes. These provide meaningful assistance to the user in processing the domain change, especially when compared to existing methods. Proposing a requirement is removed or a detail is added is far more helpful than simply stating a requirement is "impacted".

A Test Plan

This appendix shows the testplan used to test the implementation of the prototype (see also chapter 7).

A.1 Basic tests

The basic tests are meant to test if the change impact analysis rules are implemented correctly. These rules, as specified in chapter 4, consist of three types: Mapping domain changes to external inconsistencies, propagating external inconsistencies and proposing model changes corresponding to these external inconsistencies. Firstly, we test the mapping of the domain change to the external inconsistencies and the mapping of the external inconsistencies to the proposed model changes. Next, the propagation of the external inconsistencies within the model is tested. These tests are grouped by relation type.

A.1.1 Testing domain change to external inconsistency to model change mappings

For these tests, we take a model consisting of one requirement. The domain changes as listed in table 4.1 are applied to this requirement. Exception is the domain change "New requirement added", as this domain change is not be applied to a requirement. Table A.1.1 shows the tests performed and the results.

Domain change	**External inconsistency**	**Model change**	**Result**
Existing requirement removed	Requirement not in the domain, but present in the model	Requirement is removed	✓
Requirement made more specific	Requirement in the model less specific than in the domain	Detail added to requirement	✓
Requirement made more abstract	Requirement in the model more specific than in the domain	Detail removed from requirement	✓
Part removed from requirement	Requirement in the model has more parts than in the domain	Part removed from requirement	✓
New part added to requirement	Requirement in the model has less parts than in the domain	Part added to requirement	✓

Table A.1: Testing the mappings between domain change, external inconsistency and model change

All tests were succesful.

A.1.2 Testing the derivement relation

Two sets of test cases are used for the composition relation. The first uses a model consisting of two requirements R_1 and R_2, where R_1 `derived from` R_2. The tests and their results are shown in table A.2. The *or* occurrences in the table denote that the user should be presented a choice between the external inconsistency propagation possibilities.

The second model used for testing the `composed by` relation consists of three requirements R_1, R_2 and R_3, where R_1 `derived from` R_2 and R_3. The tests and their results are shown in table A.3. The *or* occurrences in the table denote that the user should be presented a choice between the external inconsistency propagation possibilities.

External inconsistency	**Expected result**	**Result**
R_1 not in domain	R_2 not in domain	✓
R_2 not in domain	R_1 not in domain *or* part of R_1 not in domain	✓
R_1 less specific than in domain	No propagation	✓
R_2 less specific than in domain	R_1 less specific than in domain	✓
R_1 more specific than in domain	R_2 more specific than in domain *or* no propagation	✓
R_2 more specific than in domain	R_1 more specific than in domain	✓
R_1 has more parts than in domain	R_2 has more parts than in domain *or* R_2 not in domain *or* no propagation	✓
R_2 has more parts than in domain	R_1 has more parts than in domain	✓
R_1 has less parts than in domain	No propagation	✓
R_2 has less parts than in domain	R_1 has less parts than in domain	✓

Table A.2: External inconsistencies with the Derived From relation (1/2)

External inconsistency	**Expected result**	**Result**
R_1 not in domain	R_2 not in domain and R_3 not in domain	✓
R_2 not in domain	Part of R_1 not in domain *or* R_1 not in domain	✓
R_1 less specific than in domain	No propagation	✓
R_2 less specific than in domain	R_1 less specific than in domain	✓
R_1 more specific than in domain	(R_2 more specific than in domain *and/or* R_3 more specific than in domain) *or* no propagation	✓
R_2 more specific than in domain	R_1 more specific than in domain	✓
R_1 has more parts than in domain	((R_2 not in domain *or* part of R_2 not in domain) *and/or* (R_3 not in domain *or* part of R_3 not in domain)) *or* no propagation	✓
R_2 has more parts than in domain	R_1 has more parts than in domain	✓
R_1 has less parts than in domain	No propagation	✓
R_2 has less parts than in domain	R_1 has less parts than in domain *or* no propagation	✓

Table A.3: External inconsistencies with the Derived From relation (2/2)

All tests were succesful. However, the current implementation does not provide assistance for the exceptions shown in table 4.2.

A.1.3 Testing composition relation

Two sets of test cases are used for the composition relation. The first uses a model consisting of two requirements R_1 and R_2, with R_1 `composed by` R_2. The tests and their results are shown in table A.2. The *or* occurrences in the table denote that the user should be presented a choice between the external inconsistency propagation possibilities.

External inconsistency	**Expected result**	**Result**
R_1 not in domain	R_2 not in domain	✓
R_2 not in domain	R_1 not in domain *or* part of R_1 not in domain	✓
R_1 less specific than in domain	No propagation	✓
R_2 less specific than in domain	R_1 less specific than in domain	✓
R_1 more specific than in domain	R_2 more specific than in domain *or* no propagation	✓
R_2 more specific than in domain	R_1 more specific than in domain	✓
R_1 has more parts than in domain	R_2 has more parts than in domain *or* R_2 not in domain *or* no propagation	✓
R_2 has more parts than in domain	R_1 has more parts than in domain	✓
R_1 has less parts than in domain	No propagation	✓
R_2 has less parts than in domain	R_1 has less parts than in domain	✓

Table A.4: Testing external inconsistency propagation for composition (1/2)

The second model used for testing the `composed by` relation consists of three requirements R_1, R_2 and R_3, where R_1 `composed by` R_2 and R_3. The tests and their results are shown in table A.5. The *or* occurrences in the table denote that the user should be presented a choice between the external inconsistency propagation possibilities.

External inconsistency	**Expected result**	**Result**
R_1 not in domain	R_2 and R_3 not in domain	✓
R_2 not in domain	R_1 has more parts than in domain	✓
R_1 less specific than in domain	(R_2 less specific than in domain *and/or* R_3 less specific than in domain) *or* no propagation	✓
R_2 less specific than in domain	R_1 less specific than in domain	✓
R_1 more specific than in domain	(R_2 more specific than in domain *and/or* R_3 more specific than in domain) *or* no propagation	✓
R_2 more specific than in domain	R_1 more specific than in domain	✓
R_1 has more parts than in domain	(R_2 not in domain *or* part of R_2 not in domain) *and/or* (R_3 not in domain *or* part of R_3 not in domain) *or* no propagation	✓
R_2 has more parts than in domain	R_1 has more parts than in domain	✓
R_1 has less parts than in domain	No propagation	✓
R_2 has less parts than in domain	R_1 has less parts than in domain	✓

Table A.5: Testing external inconsistency propagation for composition (2/2)

All tests were succesful. However, the current implementation does not provide assistance for the exceptions shown in table 4.3.

A.1.4 Testing copy relation

The tests performed are stated in table A.6, along with the results. The model used consists of two requirements R_1 and R_2, with R_1 `copy of` R_2.

External inconsistency	**Expected result**	**Result**
R_1 not in domain	R_2 not in domain	✓
R_2 not in domain	R_1 not in domain	✓
R_1 less specific than in domain	R_2 less specific than in domain	✓
R_2 less specific than in domain	R_1 less specific than in domain	✓
R_1 more specific than in domain	R_2 more specific than in domain	✓
R_2 more specific than in domain	R_1 more specific than in domain	✓
R_1 has more parts than in domain	R_2 has more parts than in domain	✓
R_2 has more parts than in domain	R_1 has more parts than in domain	✓
R_1 has less parts than in domain	R_2 has less parts than in domain	✓
R_2 has less parts than in domain	R_1 has less parts than in domain	✓

Table A.6: Testing external inconsistency propagation for copy

All tests were succesful.

B
Example model

This appendix contains the details of the requirements model used in chapter 6.

B.1 Example model

The following itemization shows the exact requirements we used. This is a subset of a much larger Course Management System requirements model. We added relations of the SysML types and added some structure. The structure itself does not directly imply a SysML relation.

The original Course Management System requirements model can be found at [CMS], we made some minor (mostly syntactic) adjustments. The requirements used here are numbered in this document as R68 to R82.

T11		The system shall allow teachers to manage dynamic course information *comp by T11.2, T11.3*
	T11.1	The system shall allow teachers to post news messages
	T11.2	The system shall allow teachers to manage the archive
		T11.2.1 The system shall allow teachers to set the visibility of archived items (enabling them to gradually expose content to students) *refines T11.2*
	T11.3	The system shall allow teachers to manage student teams *comp by T11.3.2 - T11.3.7*
		T11.3.1 The system shall allow teachers to enter grades for teams (so each team member will get that grade)
		T11.3.2 The system shall allow teachers to create new teams
		T11.3.3 The system shall allow teachers to insert students into teams
		T11.3.4 The system shall allow teachers to remove students from teams
		T11.3.5 The system shall allow teachers to delete teams
		T11.3.6 The system shall allow teachers to assign (assistant) teachers to teams
		T11.3.7 The system shall allow teachers to name and rename teams
	T11.4	The system shall provide grade statistics (averages, standard deviation, per department, per year)
		T11.4.1 The system shall enable teachers to compare grade statistics with other courses *refines T11.4*
	T11.5	The system shall allow teachers to duplicate courses and import materials from other courses into another course, but only from their own courses

Table B.1: Example requirements model (simple)

Next, we show the same requirements, but depicted as predicates and details.
The capitol letters here stand for predicates, the non-capitol for details.

T11		A, B, C, D, E, F, G, H, I, J, K *comp by T11.2, T11.3*
	T11.1	A={The system shall allow teachers to manage news messages} a={By posting news messages}
	T11.2	B={The system shall allow teachers to manage the archive}
	T11.2.1	B={The system shall allow teachers to manage the archive} b={By allowing the teacher to set the visibility of archived items} *refines 11.2*
	T11.3	C={The system shall allow teachers manage student teams}, D, E, F, G, H, I *comp by T11.3.2 - T11.3.7*
	T11.3.1	C={The system shall allow teachers to grade team members} c={By also allowing the teacher to enter grades for entire teams}
	T11.3.2	D={The system shall allow teachers to create new teams}
	T11.3.3	E={The system shall allow teachers to insert students into teams}
	T11.3.4	F={The system shall allow teachers to remove students from teams}
	T11.3.5	G={The system shall allow teachers to delete teams}
	T11.3.6	H={The system shall allow teachers to assign (assistant) teachers to teams}
	T11.3.7	I={The system shall allow teachers to name and rename teams}
	T11.4	J={The system shall store grades} j_1={And shall provide grade statistics (averages, standard deviation, per department, per year)}
	T11.4.1	J={The system shall store grades} j_1={And shall provide grade statistics (averages, standard deviation, per department, per year)} j_2={The system shall enable teachers to compare grade statistics with other courses} *refines 11.4*
	T11.5	K={The system shall allow teachers to edit courses} k_1={Also allowing teachers to duplicate courses} k_2={Also allowing teachers to import materials from other courses into another course} + k_2' k_2'={but only from their own course}

Table B.2: Example requirements model (decomposed)

Figure B.1 shows the requirements model in a graphical notation.

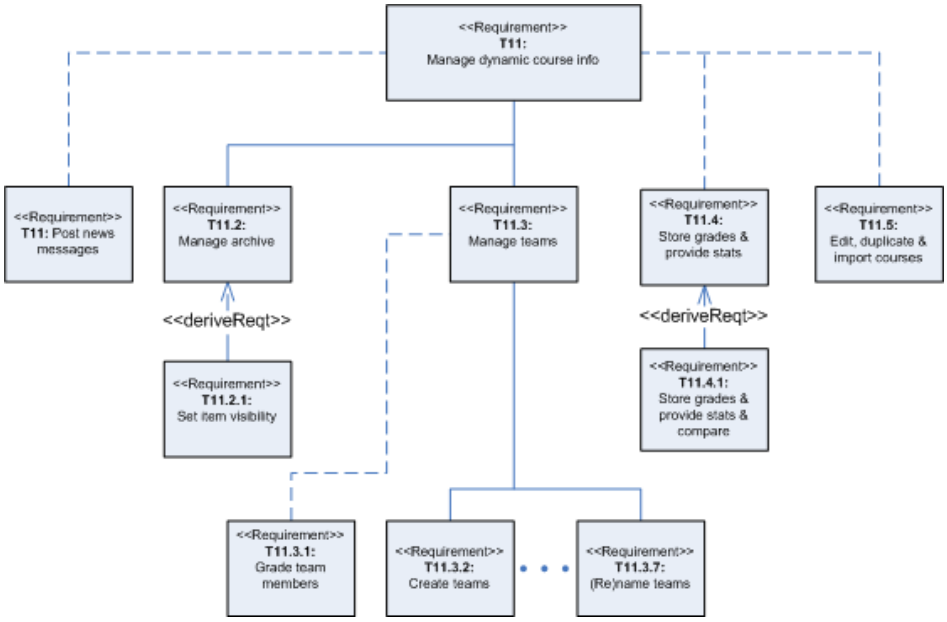

Figure B.1: Example Requirements model

The dots in the figure between T11.3.2 and T11.3.7 mark the requirements T11.3.3 to T11.3.6. The dotted lines in the figure indicate that there is some relation between the predicates of the requirements connected with this line, but this relation is not of a type specified in SysML.

References

[@-p] @-portunity. http://www.atportunity.com.

[BLOS06] L. C. Briand, Y. Labiche, L. O'Sullivan, and M. M. Sówka. Automated impact analysis of uml models. *J. Syst. Softw.*, 79(3):339–352, 2006.

[Blu] Blueprint. http://www.atportunity.com.

[CMS] Course management system. http://www.vf.utwente.nl/~goknila/sosym/Requirements_Document_for_CMS.pdf.

[FHL+98] Eckhard D. Falkenberg, Wolfgang Hesse, Paul Lindgreen, Bjrn E. Nillson, J.L. Han Oei, Colette Roland, Ronald K. Stamper, Frans J.M. Van Assche, Alexander A. Verrijn-Stuart, and Klaus Voss. Information system concepts. *The FRISCO Report*, 1998.

[GKvdB09] Arda Goknil, Ivan Kurtev, and Klaas van den Berg. Semantics of trace relations in requirements models for consistency checking and inferencing. *Software and Systems Modeling*, 2009.

[JL04] P. Jnsson and M. Lindvall. *Engineering and Managing Software Requirements*. 2004.

[KS98] Gerald Kotonya and Ian Sommerville. *Requirements Engineering: Processes and Techniques*. Wiley, 1998.

[LS98] M. Lindvall and K. Sandahl. How well do experienced software developers predict software change? *The Journal of Systems and Software*, 43:19–27, 1998.

[OMG02] OMG Object Management Group. Metaobject facility (mof) specifcation. Technical report, http://www.omg.org/cgi-bin/doc?formal/2002-04-03, April 2002.

[OMG03] OMG Object Management Group. Uml 2.0 ocl specification. Technical report, http://www.omg.org/docs/ptc/03-10-14.pdf, October 2003.

[OMG06] OMG Object Management Group. Metaobject facility (mof) specifcation. Technical report, http://www.omg.org/spec/MOF/2.0, January 2006.

[OMG07a] OMG Object Management Group. Omg systems modeling language (omg sysml™), v1.0. Technical report, http://www.omg.org/spec/UML/2.3/, September 2007.

[OMG07b] OMG Object Management Group. Omg unified modeling language (omg uml), infrastructure, v2.1.2. Technical report, http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF, November 2007.

[Req] Requisitepro. http://www-01.ibm.com/software/awdtools/reqpro/.

[RI06] S. Ramzan and N. Ikram. Requirement change management process models: Activities, artifacts and roles. In *Multitopic Conference, 2006. INMIC '06. IEEE*, pages 219 –223, dec. 2006.

[SSB93] Anthony M. Starfield, Karl Smith, and Andrew L. Bleloch. *How to Model It: Problem Solving for the Computer Age*. McGraw-Hill, Inc., New York, NY, USA, 1993.

[vL09] A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.