SINTEF A6901

REPORT

ECMDA Traceability Workshop (ECMDA-TW) 2008 Proceedings

Editors: Jon Oldevik, Gøran K. Olsen, Tor Neple

Collaborative and Trusted Systems

Information and Communication Technology

June 2008

ECMDA Traceability Workshop (ECMDA-TW) 2008 Proceedings

June 12th 2008, Berlin, Germany

This workshop was organised in collaboration with the European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA)

Organisers:
Jon Oldevik, Gøran K. Olsen, Tor Neple, SINTEF
{jon.oldevik / goran.k.olsen / tor.neple} at sintef.no
Richard Paige, University of York, UK
(paige at cs.york.ac.uk)

ISBN 978-82-14-04396-9

SINTEF ICT Address: NO-7465 Trondheim, NORWAY Location: Forskningsveien 1 Telephone: +47 22 06 73 00 Fax: +47 22 06 73 50 Enterprise No.: NO 948 007 029 MVA	**SINTEF REPORT** TITLE **ECMDA Traceability Workshop Proceedings 2008** AUTHOR(S) Jon Oldevik, Gøran K. Olsen, Tor Neple, Richard Paige CLIENT(S) ECMDA Traceability Workshop 2008

REPORT NO. SINTEF A6901	CLASSIFICATION Open	CLIENTS REF. ECMDA Traceability Workshop 2008	
CLASS. THIS PAGE	ISBN 978-82-14-04396-9	PROJECT NO. 90B234	NO. OF PAGES/APPENDICES 120 / 0
ELECTRONIC FILE CODE ECMDA-TW Proceedings		PROJECT MANAGER (NAME, SIGN.) Tor Neple [illegible]	CHECKED BY (NAME, SIGN.) Arnor Solberg [illegible]
FILE CODE	DATE 2008-06-01	APPROVED BY (NAME, POSITION, SIGN.) Bjørn Skjellaug, Research Director [illegible]	

ABSTRACT

This report contains the proceedings of the Fourth ECMDA Traceability Workshop, arranged in Berlin, Germany 2008 together with the ECMDA 2008 conference. The papers within target various aspects of traceability in model-driven development.

KEYWORDS	ENGLISH	NORWEGIAN
GROUP 1	ICT	IKT
GROUP 2	Information Systems	Informasjonssystemer
SELECTED BY AUTHOR	Traceability	Sporbarhet

Table of contents

ISBN 978-82-14-04396-9

ISBN 978-82-14-04396-9

Introduction

This is the fourth Traceability Workshop organised in the context of the ECMDA conference series. For this year's workshop we received 15 papers and accepted 10 of them, after detailed review and revision. The focus of the papers spanned from the quite theoretical to implemented solutions for practical problems.

We trust that the workshop papers, and the presentation of these, act as catalyst for constructive discussion both on the theoretical and practical aspects of traceability in model driven engineering. The record number of submitted and presented papers indicates that the topic of traceability still is a focus within the community. From last year's workshop the message taken away by the organising committee was that industrial adoption of traceability techniques and theories was hindered by lack of mature tools and standardisation. During this year's workshop we would like to revisit these topics, among others, to see if there has been improvement on these areas.

With this we wish you all a fruitful workshop.

Acknowledgment: The organisation of the ECMDA-TW workshop was made possible through the MODELPLEX[†] project (IST Project 34081).

– ECMDA Traceability Workshop Organising Committee, June 2008.

[†] This work is a result from the MODELPLEX project co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (http://www.modelplex-ist.org/). Information included in this document reflects only the authors' views. The European Community is not liable for any use that may be made of the information contained herein.

ISBN 978-82-14-04396-9

Program Committee

ISBN 978-82-14-04396-9

A Traceability Engine Dedicated to Model Transformation for Software Engineering

Bastien Amar, Hervé Leblanc, and Bernard Coulette

IRIT, Université Paul Sabatier,
118 Route de Narbonne,
F-31062 Toulouse Cedex 9
{amar,leblanc,coulette}@irit.fr

Abstract. This paper deals with the use of a model transformation traceability engine in a model-driven process. We propose to use traces of model transformations for the visualization and debugging of an example of model refactoring transformation. We start off with the transformation Java code and analyse the trace graph generated by the framework during the transformation execution. A short description of the framework and its functionalities is presented. We compare our work with frameworks already proposed to manage traceability in a MDE context.

1 Introduction

With the advent of languages and tools dedicated to model-driven engineering (*e.g.*, ATL[1], Kermeta[2], EMF[3]), as well as reference metamodels (MOF, Ecore), model-driven development processes can be used easily. As a result, recurring problems linked to software production are emerging in this new context of development. One of those issues concerns traceability, which comes especially from requirements engineering. In a MDE context, traceability is achieved by the definition and maintaince of relationships between artifacts involved in the software-engineering life cycle during system development [1]. This paper aims at describing the use of a model transformation traceability engine in a model-driven process.

To illustrate the presentation, we provide directly a case study. We consider an imperative java transformation using the EMF platform and we deduce a graph trace transformation thanks to a traceability engine. We present some hypotheses about the use of the graph for the visualization and debugging of a model transformation (section 2). Then we present the main features of our engine which is an Eclipse plug-in. Its main particularities are (i) the use of the aspect oriented programming paradigm in order to isolate trace generation from code transformation, (ii) a composite trace meta-model which allows to modularize link sets at different granularity levels (section 3). We make a comparison

[1] http://www.eclipse.org/m2m/atl/.
[2] http://www.kermeta.org.
[3] http://www.eclipse.org/modeling/emf/.

ISBN 978-82-14-04396-9

with other works that deal with change tracking and use of traces to facilitate the writing of new transformation types (section 4). Finally, we conclude and give some perspectives about software engineering for model transformations (section 5).

2 Case Study

The case study presented in this section will help us to present the issues and an example of results we obtain with our traceability platform. This is divided into two parts: the first is an example of model transformation written in Java EMF, the second is a representation of the transformation process in the form of annotated graph. This graph is deduced from automatically generated traces by a model to text transformation. Here, a trace model that conforms with a composite meta model is transformed to a .dot notation which is an entry to a graph visualization tool. This graph will provide us better understanding of the transformation process, and as a result, can be a valuable aid in debugging.

2.1 Transformation Code

Figure 1 shows the source code of a Java transformation in the considered environment. This transformation is endogenous and horizontal according to the classification criterias of [2]. It's an abstraction at design model level of refactoring code transformations proposed by Martin Fowler [3]. We named this class `GeneralizationProcess` because the main method (`run()`) would be executed at the end of a design stage for eliminating redundancies due to different points of view or to a collaborative process development. The `run` method initializes context and launches backup of the result models. The designer of a transformation has the possibility to encapsulate code subject to traces generation *via* a call to the `transform` method. During a transformation execution, all events affecting a modeling element are used to name the corresponding traceability link.

Now consider the two methods named respectively `extractSuperClass` and `pullUpField`. Their operational semantics are equivalent to Martin Fowler's refactoring motivations:

- Extract Superclass: you have two classes with similar features, then create a superclass and move the common features to the superclass.
- Pull up Field: two subclasses have the same field, then move the field to the superclass.
- Extract Superclass calls Pull up Field: one by one, use Pull up Field to move common elements to the superclass.

Finally, the dependant platform code in Java EMF is found in the following low-level operations:

ISBN 978-82-14-04396-9

```
public class GeneralizationProcess {

  public void run() {
    Resource model = Util.load(path);
    EPackage ep=(EPackage)model.getContents().get(0);
    EList<EClassifier> classes = ep.getEClassifiers();
    transform(classes);
    Persistent.save("./model/Result.ecore",ep);
  }

  public void transform(EList<EClassifier> classes) {
    for (int i = 0; i<classes.size(); i++) {
      for (int j=i+1; j<classes.size(); j++){
        EClass a = (EClass) classes.get(i);
        EClass b = (EClass) classes.get(j);
        if (needRefactoring(a,b)) {
          classes.add(extractSuperClass(a,b));
        }
      }
    }
  }

  public EClass extractSuperClass(EClass a, EClass b) {
    EClass result = EcoreFactory.eINSTANCE.createEClass();
    result.setName(a.getName() + "␣^␣" + b.getName());
    EList<EAttribute> aAttribute = a.getEAttributes();
    EList<EAttribute> bAttribute = b.getEAttributes();
    for (int i=0; i<aAttribute.size(); i++ ){
      for (int j=0; j<bAttribute.size(); j++ ){
        if (aAttribute.get(i).getName().equals(bAttribute.get(j).getName())){
          PullUpField(aAttribute.get(i), bAttribute.get(j),result);
        }
      }
    }
    result.setAbstract(true);
    addInheritanceLink(a, result);
    addInheritanceLink(b, result);
    return result;
  }

  public EAttribute PullUpField(EAttribute a1,
                                EAttribute a2,
                                EClass target) {
    EAttribute result = EcoreFactory.eINSTANCE.createEAttribute();
    result.setName(a1.getName());
    result.setEType(a1.getEType());
    a1.getEContainingClass().getEStructuralFeatures().remove(a1);
    a2.getEContainingClass().getEStructuralFeatures().remove(a2);
    target.getEStructuralFeatures().add(result);
    return result;
  }

  private void addInheritanceLink(EClass subclass, EClass superclass) {
    subclass.getESuperTypes().add(superclass);
  }
}
```

Fig. 1. Generalization process class

ISBN 978-82-14-04396-9

– The creation of a new modelling artefact is translated by a call to a specific method on a unique dedicated Factory class: `createE<TypeArtefact>`. The creation of an artefact does not imply the persistence of it in results model.
– The addition or removal of a model artefact is translated by the same operation on a list of artefacts contained by the appropriate container of model elements, for example: the addition of an attribute to a class container or the addition of a class to a package container.

2.2 Transformation Traces Representation

We execute our transformation on a source model that contains two classes `A` and `B` with a common boolean attribute named `attrA`. Then, the target model contains a common super class `A` ∧ `B` that factorizes this attribute. The figure 2 represents a trace graph corresponding to the execution of the transformation. This graph was generated thanks to our traceability engine.

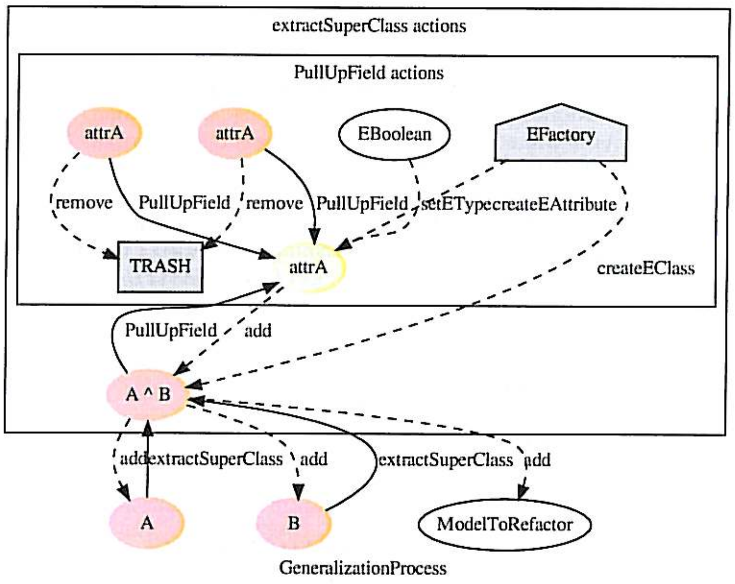


Fig. 2. Trace graph generated by ETraceTool

The nested traces are represented by boxes. The method `transform` has been declared in `GeneralizationProcess` class which is the name of the main

ISBN 978-82-14-04396-9

composite trace link. The method calls graph is isomorphic to the nested traces graph, and the box named ExtractSuperClass includes the box named PullUpField. Lines represent trace links from source to target element of a transformation. Solid lines represent high level transformation links (of user-specific operations) and dashed lines represent low level transformation links (of API-specific operations). For example, classes `A` and `B` are source elements of the high-level transformation extract super class, while classe `A` ∧ `B` is source element of two low-level transformations which consist in adding inheritance links to the two classes `A` and `B`. We can deduce easily that the attribute named **`attrA`** is the factorized common feature. We can remark that attributes **`attrA`** in classes `A` and `B` are not lost. They are saved in a generic container named Trash.

This annotated graph can be considered as a static visualization of a transformation. How can it be used as a debugging transformation tool ? Due to the composite structure of our trace meta model, we can generate a sub graph directed by the scope of a transformation method or by elements model type involved in a transformation process.

3 Realization

In this section, we make a brief description of ETraceTool, a traceability platform dedicated to imperative model transformations. The platform is encapsulated in an Eclipse plug-in. It permits us to generate all links of traceability during a model transformation. Transformations must be written in Java using the EMF API [4]. EMF is an Eclipse tool based on Ecore metamodel, which is equivalent to the MOF provided by OMG. It's the core of many meta modeling tools (Kermeta from INRIA, SmartQVT [5] from OrangeLabs and the open source project TopCased [6]).

A prototype of our plugin is available for the SmartQVT platform: SmartQVT is an implementation of the OMG standard "QVT operational" and compiles its transformations to Java/EMF. So, we easily adapted our plug-in to this language.

Below, we present the main features of our plug-in:

- The trace generation code is not intrusive in the transformation code.
- Trace generation is explicitly activated by the designer of the transformation.
- Trace models are isolated from source and target models involved in transformation.
- Trace models can be used at different levels of granularity. In the example, creation of an attribute can be considered as a sub-level granularity link for the creation of a class link.

The figure 3 represents the platform architecture. During the transformation, we catch categorized events thanks to aspect oriented programming [7], and the `Tracer` aspect generates a trace model conform to a boxed links metamodel. At the end, the trace model can be serialized in a XMI file or transformed to dot language *via* a model-to-text transformation [8]. Other transformations can be

ISBN 978-82-14-04396-9

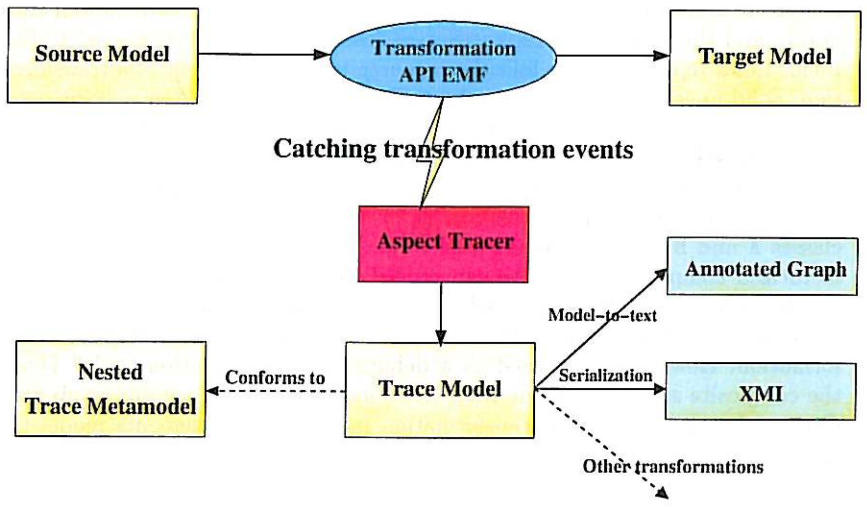


Fig. 3. ETraceTool architecture

conceived from the model in memory or from the serialized model. We focus now on the two original points of our approach: the use of a single aspect for isolating the traceability code and our metamodel composite structure.

3.1 Catching transformation events

Aspect-oriented programming is a programming paradigm that allows to separate implementation of a cross-cutting concern from application code. The principle is to code separately the issues, and define integration rules for combining the different issues to create the final system. Compared to object-oriented programming, this paradigm allows to encapsulate behavior that affects multiple classes in a reusable module. In our case, the generation of traces is a concern, and the transformation is the core of the application. AspectJ[4] is used to implement our prototype.

All that remains is to categorize operations used to manipulate models in the EMF platform. We associate the corresponding pointcut pattern to each of them. They are used to identify events to be catched by a regular expression. During the transformation process, the categorized events are catched and the trace model is built in the following manner [9]:

- Before the execution of the method intercepted, links are built and their sources are affected.
- After the execution of the method, their targets are affected.

[4] http://www.eclipse.org/aspectj/

ISBN 978-82-14-04396-9

3.2 A nested trace metamodel

A traceability platform keeps information on the evolution of the models during the various undergoing transformations. A trace model is associated with each execution of a transformation. The definition of a trace metamodel allows us to structure the traces generated by our platform. Our work deals with model transformations, and we will study only metamodels dedicated to this activity.

Several trace metamodels have already been proposed [10, 11]. They support most of the transformations envisaged in a MDE process [2].

The core of the trace notion was presented in [10]: a trace link is composed by a set of sources and a set of targets elements. It's adapted to declarative model transformation languages, such as ATL, for which it was designed.

The first extension of the trace notion is to consider a trace as a set of bipartite graphs with a common intersection [11]. A step artefact permits to manage chain-transformation - several transformations performed successively on models. The difference is the trace multiplicities: a trace has an unique source and target.

Our metamodel (Figure 4) extends the one proposed in [11]. It's useful, for imperative as well as declarative transformations, to have a multiscaled trace. The fact that an operation transformation can call another one (or that the rules can trigger other rules) creates levels of nesting that it's useful to be able to represent. That's why the composite pattern is applied on the links [12].

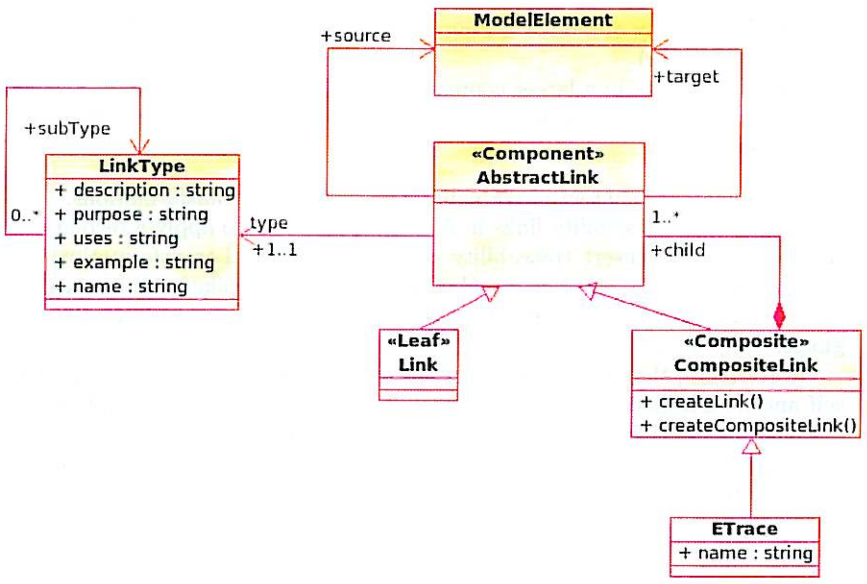


Fig. 4. The nested trace metamodel

ISBN 978-82-14-04396-9

We have added the concept of link types (LINKTYPE) to allow the user to reason on links [13]. We also added composite links (COMPOSITELINK). A trace (ETRACE) is a kind of COMPOSITELINK, *ie* composed of multiple links (ABSTRACTLINK) that reference two model elements (SOURCE and TARGET).

In our case, the composite link allows us to separate low-level operation (creation, deletion...) from high-level operation (a refactoring transformation), while we maintain a composite link between both. With the composite pattern, we also support chain-transformation on models.

A link has necessarily one source and one destination. If we want to trace an element deletion, which is a low-level operation, we create a generic container to store deleted items. The link target will be this container, named "Trash" on figure 2. For an element creation, we proceed in the following manner: the factory will be the source and the newly created element the destination.

4 Related works

In software engineering, traceability has two main semantics, depending on the context [14]:

- Traceability in requirements engineering is tracking a requirement from its expression to its implementation. Traceability is strongly influenced by the originators of traceability : the requirements management community [1].
- Traceability in model-driven development can be subdivided in two categories:
  - Traceability of models during a transformation (or traceability of transformations)
  - Traceability in a larger context of model-driven development. This part explores metamodelling of traces and their potential uses in a MDE tools suite.

Our work is in the context of traceability during model transformations.

To generate traceability links in ATL, [10] proposes to apply a second order transformation to insert traceability rules in the original transformation. This transformation of a program can be seen as a precompilation or as a meta-transformation. However, this approach is only applicable to declarative language.

In Kermeta, the programmer must enter the code for traces generation himself and no automation is proposed [11]. The approach has been implemented, and the perspective suggests a future management of markers in the code to automate traces generation. Our approach is different: we're trying to avoid the presence of traceability information in the original transformation to not pollute it.

ISBN 978-82-14-04396-9

5 Conclusion

In this paper we have presented our traceability platform adapted to an imperative language. It works and is available as an Eclipse plugin. We propose a trace visualization in the form of an annotated graph which allows us to analyse a model transformation. Now that we have a working traceability framework, we plan to implement the "undo / redo" of an endogenous transformation. The use of the traceability during a MDE process may reveal new problems, such as the decrease of transformation performances, the management of consistency between traceability links and references to source or target elements, the possibility for the designer to choose specific elements or events to trace.

For the debugging problem, the creation of a dynamic trace representation system may allow the user to navigate easily through the different levels of trace and to find problems more efficiently.

Finally, we would like to focus our research work on transformations engineering: use traces for new transformations, such as incremental transformations or refinement injection.

Acknowledgement

This work is partially supported by the TOPCASED and DOMINO projects.

References

1. Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., Shaham-Gafni, Y.: Model traceability. IBM Syst. J. **45**(3) (2006) 515–526
2. Mens, T., Gorp, P.V.: A taxonomy of model transformation. Electr. Notes Theor. Comput. Sci. **152** (2006) pp. 125–142
3. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional (June 1999)
4. Budinsky, F., Grose, T., Steinberg, D., Ellersick, R., Merks, E., Brodsky, S.: Eclipse Modeling Framework: a developer's guide. Addison-Wesley Professional (2003)
5. http://smartqvt.elibel.tm.fr/: An open source model transformation tool implementing the MOF 2.0 QVT-Operational language.
6. http://www.topcased.org/: Toolkit in open-source for critical application & systems development
7. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: Proceedings European Conference on Object-Oriented Programming. Volume 1241. Springer-Verlag, Berlin, Heidelberg, and New York (1997) 220–242
8. http://www.graphviz.org: Graphviz - graph visualization software
9. Amar, B., Falleri, J.R., Huchard, M., Nebut, C., Leblanc, H.: Un framework de traçabilité pour des transformations à caractère impératif. In: Conférence sur les Langages et Modèles à Objets (LMO), Montréal (Canada), 05/03/08-07/03/08, http://www.cepadues.com/, Cépaduès Editions (février 2008) 141–154
10. Jouault, F.: Loosely coupled traceability for ATL. In: ECMDA-Traceability Workshop, Nuremberg. (2005) 29 – 36

ISBN 978-82-14-04396-9

11. Falleri, J.R., Huchard, M., Nebut, C.: Towards a traceability framework in Kermeta. In: ECMDA-TW 2006 Proceedings, Bilbao, July 11th 2006. (2006) 31 – 40
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (January 1995)
13. Limòn, A.E., Garbajosa, J.: The need for a unifying traceability scheme. In: ECMDA Workshop on traceability. November 8th 2005, Nuremberg Germany. (2005) 47 – 56
14. Galvao, I., Goknil, A.: Survey of traceability approaches in model driven engineering. In: the Eleventh International IEEE EDOC Conference (EDOC 2007), IEEE Computer Society Press (2007)

ISBN 978-82-14-04396-9

Towards Rigorously Defined Model-to-Model Traceability

Nicholas Drivalos[1,2], Richard F. Paige[1], Kiran J. Fernandes[2], Dimitrios S. Kolovos[1]

[1] Department of Computer Science, University of York
{nikos, paige, dkolovos}@cs.york.ac.uk
[2] The York Management School, University of York
kf501@.york.ac.uk

Abstract. A Model Driven Engineering process typically involves models expressed in different modelling languages that capture different views of the system under development. To enhance automation, consistency and coherency, establishing and maintaining semantically rich traceability links between model elements that belong to different models used throughout the process is of paramount importance. In this paper we propose a rigorous approach to defining such semantically rich traceability links between models expressed in diverse modelling languages using case-specific traceability metamodels and demonstrate the practicality and usefulness of our approach using a concrete example.

1 Introduction

Model-Driven Engineering (MDE) is an approach to software development where the primary focus is on models, as opposed to source code. A model describes certain views of the software system and its environment at a certain level of abstraction. MDE uses models to represent all artefacts that are involved in the software development process, such as requirements, software components or system documentation. Usually, models are described by different languages and they can be refined, evolved into a new version, and used to produce other models or even executable code. The ultimate goal of MDE is to raise the level of abstraction, and to develop and evolve complex software systems by manipulating models [9].

In a typical MDE process, many different and heterogeneous modelling artefacts are produced. This poses challenges to the traceability of the various models elements, i.e. the ability to establish, represent and update relationships among the various artefacts developed during the software development life-cycle. Traceability is considered as a necessary system characteristic since it underpins software management, software evolution and validation [14].

In this paper we present an approach to specifying and capturing strongly typed and semantically rich traceability links between models that conform to potentially different metamodels. In our approach we use a dedicated traceability

ISBN 978-82-14-04396-9

metamodel to specify type safe traceability links of interest, as well as a set of inter-model constraints for verifying the validity of the established links.

The rest of the paper is organised as follows: In Section 2, we define the main concepts, which are met throughout this paper, while in section 3 we present the two approaches to storing and managing traceability information, as well as the advantages and disadvantages of each approach. In Section 4, we propose the main contribution of this paper, which is the use of case-specific traceability metamodels and inter-model constraints, that define strongly typed and semantically rich traceability links. In Section 5, we present a concrete example that demonstrates the practicality and usefulness of our approach, while in section 6 we provide a discussion on related work and compare to our approach. Finally, in Section 7 we conclude and identify interesting further work on the subject.

2 Background

Traceability is defined in the IEEE Standard Glossary of Software Engineering Terminology [16] as follows:

> The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match.

The above definition exhibits a strong influence from the originators of the traceability concept, the requirements management community. However, for the purpose of traceability in the context of MDE a broader definition is required. Such a definition is provided in [2], where traceability is considered as any relationship that exists between artefacts of the software development life-cycle, such as mappings generated as a result of transformations or relationships that are computed based on existing information (e.g. dependency analysis).

Following [10], there are two strategies to storing and managing model-to-model traceability information. In the first one, the traceability information is embedded in the models they refer to, while in the second strategy the traceability information is stored separately from the models.

2.1 Intra-Model Storage of Traceability Links

Under this strategy, the traceability information is stored within the artefact they refer to in the form of model elements or model element attributes, such as tags and properties. Despite its simplicity and human friendliness, keeping such information with the artefacts can be problematic for several reasons. If the link is directed and stored in the source model only, it is not visible from the target model. On the other hand, if the traceability information is stored in both models, then this information must be maintained consistent, thus the burden

ISBN 978-82-14-04396-9

of maintaining consistency is introduced every time a change occurs [2]. In addition, embedding traceability information inside a model causes *"pollution"* [10], i.e. the inclusion of elements in the model of secondary importance. Such an inclusion can render a model overcrowded and can make it difficult to understand and maintain. Finally, the issue of *uniformity* arises in this approach [1]. In an MDE environment, it is common that models have their own representations and semantics. Hence, it is very difficult to distinguish the traceability information from the other model artefacts. As a result, automated analysis of traceability information becomes very challenging. The main approaches falling under this strategy utilise mainly language specific constructs. For example, specific types of traceability links are represented in UML diagrams by using stereotyped dependencies, such as $\ll refines \gg$ [8].

2.2 External Storage of Traceability Links

Under this strategy, traceability information is stored separately from the models they refer to in a separate model. Constructing such models has two clear advantages. First, the source and target models remain "clean", since the traceability links are stored in a separate model, whose concern is to capture this kind of information. In this way, the aforementioned "pollution" is avoided. In addition, storing traceability links in a model who conforms to a metamodel with clearly defined semantics makes automatic analysis by tools much easier. A prerequisite for storing traceability links externally from the models they refer to, is that the various model elements have unique identifiers, so that the related traceability links can be resolved unambiguously [10]. For example, such a mechanism is provided by MetaObject Facility (MOF) [15] and Eclipse Modeling Framework (EMF) [6] in the form of a *xmi.id* identifier. In our view, storing traceability links in separate models is more preferable than embedding the traceability information in the models they refer to. This is because in addition to the aforementioned advantages, this strategy is able to capture both intra-model as well as inter-model links.

3 A Generic Approach to Inter-Model Traceability Establishment

In the general case we need to establish links between elements belonging to a number of models that potentially conform to diverse metamodels. Also, several types of traceability links linking different types of model elements may need to be captured. As discussed, traceability links should not be captured by constructs internal to the models they refer to, to reduce "pollution". Instead, they should be captured in the form of a separate model. This model that contains the traceability links must conform to a metamodel. There are two alternatives: use a general-purpose traceability metamodel or use multiple case-specific traceability metamodels.

ISBN 978-82-14-04396-9

3.1 General-Purpose Traceability Metamodel

In this case, a generic metamodel that enables capturing relationships between any types of model elements is used. In this metamodel, a traceability link can connect any number of elements, of any type in any model. Such a metamodel is the *Unified Traceability Scheme* developed in [12]. The main advantages of a general-purpose metamodel are simplicity and uniformity (as all traceability models conform to the same metamodel) which creates potentials for enhanced tool-interoperability as tools will be able to import, export and manage traceability in a common format. On the other hand, such a general purpose metamodel does not capture case-specific strongly typed traceability links with rigorously defined semantics and constraints, and thus allows establishment of potentially illegitimate traceability links. For example, in the case we want to represent traceability links between a class diagram and a relational database model and we know that links exist between classes of the former model and tables of the later, a generic traceability metamodel allows the establishment of possibly illegitimate links, such as class-column links. Provision of extension mechanisms along with the general-purpose traceability metamodel is an approach often used to allow better support for case-specific requirements. Examples of this approach can be efficient in tackling the aforementioned issues [3, 13]. However, they still lack the efficiency of case-specific metamodels to capturing case specific information and semantics.

3.2 Case-Specific Traceability Metamodel

In this case, for each traceability scenario, a case-specific traceability metamodel is defined. This traceability metamodel captures case-specific strongly typed traceability links with well-defined semantics that potentially include correctness constraints that extend beyond simple type conformance. Apparently, such a metamodel can capture more case-specific information and semantics and, due to its strongly typed nature and the attached constraints, restricts users and tools to establishing legitimate traceability links only. By contrast, a case-specific traceability metamodel requires some effort to be spent for its construction and also tools that support different traceability metamodels shall not be able to directly communicate with each other. Nevertheless, the process of establishing an explicit traceability metamodel is to our view beneficial in the long term as it involves the engineers in a cognitive process through which the conceptual correspondences between the respective metamodels that are involved in the traceability scenario are enhanced. Also, with the advent of model management technologies (e.g. model transformation languages) transforming between different well-defined traceability metamodels is expected to be a straightforward process to automate.

ISBN 978-82-14-04396-9

4 Establishing a Strongly Typed, Semantically Rich Traceability Metamodel

In this paper we propose an approach for capturing and representing strongly typed and semantically rich traceability links. This approach involves establishing a traceability metamodel that defines strongly typed traceability links and a set of constraints that express validity requirements that can not be captured by the metamodel itself.

To be strongly typed, the traceability metamodel needs to explicitly refer to types of elements defined in other metamodels. For example, consider that we need to define a traceability metamodel that enables establishment of simple traceability links between instances of A (from MMa) and instances of B (from MMb), but no links between two instances of A or two instances of B. To capture such a metamodel, the underlying modelling technology must not consider each metamodel as a closed space - but instead allow inter-metamodel references. An exemplar technology that supports inter-metamodel references is the Eclipse Modeling Framework (EMF) [6].

Although a metamodel captured using a modelling technology that allows inter-metamodel references can enforce type safety, there are often additional requirements that need to be specified, and which the traceability metamodel cannot capture by it self. For instance, in the previous example, an additional requirement may be that each instance of A from MMa can only be linked to no more than one instances of B in MMb. To specify such constraints, a constraint specification language that can express constraints spanning over elements belonging to a number of models of potentially different metamodels is required. The Object Constraint Language (OCL) [17] currently lacks such capabilities as it does not provide constructs for expressing cross-model constraints. Exemplar constraint languages that support establishing cross-model constraints include the Epsilon Validation Language (EVL) [11] and the XLinkit [4] toolkit.

The combination of a strongly typed traceability metamodel with verifiable inter-model constraints restricts users and tools to establishing and maintaining only meaningful traceability links, which can be automatically validated to discover potential omissions and inconsistencies. Such issues can arise either during the establishment of the traceability links or later on in the lifecycle of the models among which traceability links have been established. In the next section, we demonstrate the practicality and usefulness of our approach using a concrete example.

5 Example

In this section, we present the proposed methodology using a concrete example. In our example, we use two simple metamodels, the *ClassMetamodel* and the *ComponentMetamodel*, which are illustrated at the top and bottom of Figure 1 respectively. Our aim is to capture traceability links between instances of Package from the *ClassMetamodel* and Component from the *Component metamodel*,

ISBN 978-82-14-04396-9

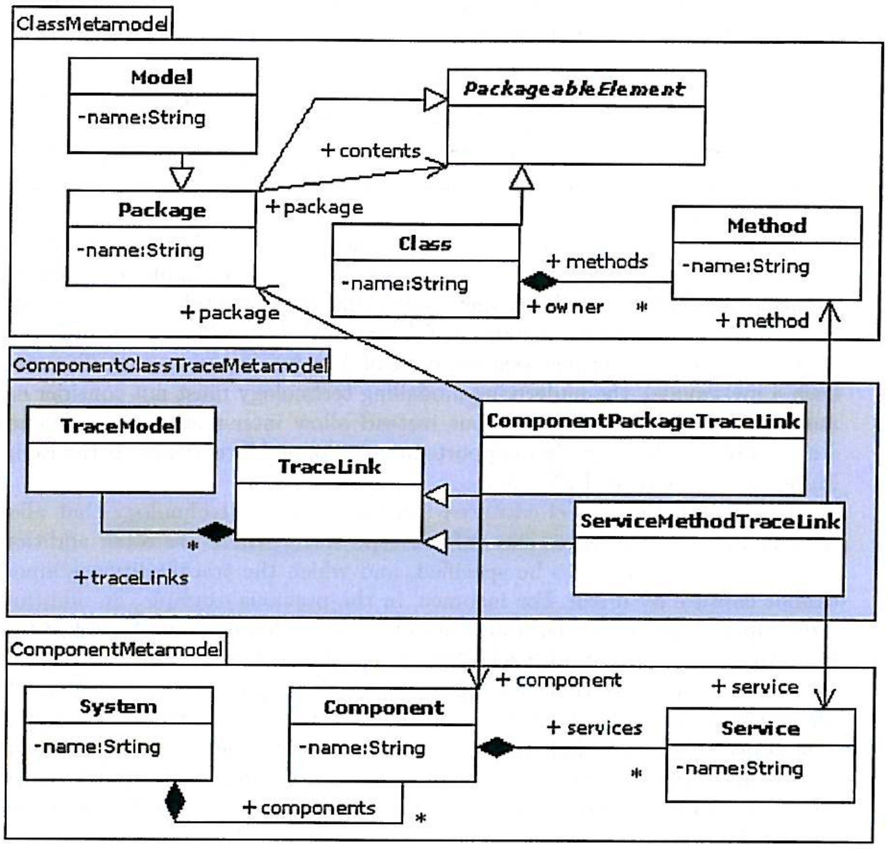


Fig. 1. The ClassMetamodel, ComponentMetamodel and ClassComponent-TraceLinkMetamodel metamodels

and links between instances of Method from the *ClassMetamodel* and Service from the *ComponentMetamodel*. Furthermore, the following exemplar constraints must be satisfied by the trace model (*ComponentClassTraceModel*):

- **(C1)** For each Service in *ComponentModel* there is exactly one *ServiceMethod-TraceLinkTrace* in the *ComponentClassTraceModel* that links it with a Method in the *ClassModel*
- **(C2)** If a Service in the *ComponentModel* is linked to a Method in the *ClassModel* via a *ServiceMethodTraceLink*, then the component in which the service is defined must also be linked with the *Package* in which the *Class* that contains the Method is defined.

The first step of the solution is to define the *ClassComponentTraceMeta-model* case-specific traceability metamodel that is displayed as a shaded pack-

ISBN 978-82-14-04396-9

age in the middle of Figure 1. The metamodel specifies a *TraceModel* container, an abstract *TraceLink* class, and the *ComponentPackageTraceLink* and *ServiceMethodTraceLink* classes that extend *TraceLink*. The *ComponentPackageTraceLink* defines two references: the package reference that is of type *Package* and the component reference which is of type *Component*, from the respective metamodels. Similarly, the *ServiceMethodTraceLink* defines the service and method references which are of type *Service* and *Method* respectively. By specifying explicitly the supported traceability links, we preempt- establishment of illegitimate links (e.g. tracing a *Component* to a *Method*).

However, the metamodel cannot enforce the additional constraints C1, C2 discussed above by itself. To impose such constraints, we use the Epsilon Validation Language [11], a constraint language that is capable of expressing constraints over multiple models of different metamodels simultaneously.

In Listing 1.1 constraint C1 applies to all instances of *Service* in the *ComponentModel* and in line 5 it calculates the number of *ServiceMethodTraceLink* in the *ComponentClassTraceModel* that link the *Service* with a *Method* from the *ClassModel*. In line 6 it returns true if exactly one trace link is found and false otherwise. Then in line 8 it reuses the *count* variable calculated in the *check* part of the constraint to generate an appropriate error message according to whether zero or more than one trace links have been found.

Listing 1.1. Constraint C1 expressed in EVL

```
1  context ComponentModel!Service {
2   constraint C1 {
3    check {
4     var count := ComponentClassTraceModel!ServiceMethodTraceLink.
5      all.select(sml|sml.method = self).size();
6     return count = 1;
7    }
8    message {
9     if (count = 0) {
10     return 'Service ' + self.name + ' does not trace to a method';
11    }
12    else {
13     return 'Service ' + self.name + ' traces to many methods';
14    }
15   }
16 }
```

In Listing 1.2 constraint C2 is evaluated against all instances of ServiceMethodTraceLink in the ComponentClassTraceModel and checks that there exists at least one *ComponentPackageTraceLink* that links the component in which its *service* is defined with the Package in which the class that contains the *method* is defined.

Listing 1.2. Constraint C2 expressed in EVL

```
1  context ComponentClassTraceModel!ServiceMethodTraceLink {
2   constraint C2 {
```

ISBN 978-82-14-04396-9

```
    check : ComponentClassTraceModel!ComponentPackageTraceLink.all
      .exists(cpl|self.service.component = cpl.component
        and self.method.owner.container = cpl.package)
    message : 'The package in which method '
      + self.method.name + ' is defined does not trace '
      + 'to the component in which service '
      + self.service.name + ' is defined'
  }
}
```

In this example we have demonstrated our approach on a simple but representative example. To establish rigorous traceability links between elements of the exemplar *ComponentMetamodel* and *ClassMetamodel* metamodels we have introduced a new metamodel (*ComponentClassTraceMetamodel*) in which every legitimate type of traceability link is represented as a separate metaclass that contains references to the types of elements it can link. Moreover, we have demonstrated that additional constraints which cannot be captured by the traceability metamodel are necessary for rigorously specifying legitimate traceability links, and shown how such constraints can be captured using a constraint language (EVL) that supports accessing multiple models simultaneously.

6 Related Work

6.1 Atlas Model Weaver

AMW, the Atlas Model Weaver [13], is a tool created by INRIA as part of the ATLAS Model Management Architecture . Its primary goal is to capture and to store links between models. This information is stored in a model, which is called *weaving model*. This metamodel is very flexible and may be extended to add additional mapping semantics. The process of creating the weaving model can be manual or semi-automatic. AMW provides a base weaving metamodel enabling to create links between model elements and associations between links. The main difference of AMW and the approach proposed in this paper is the fact that AMW treats the weaving model as a closed space and does not allow inter-metamodel references, while our approach focuses on those references since they provide the basis for a more semantically rich weaving metamodel.

6.2 C-SAW

The Constraint-Specification Aspect Weaver (C-SAW) is a general transformation engine for manipulating models and is a plug-in for GME [19]. This approach is strongly influenced by the Aspect Oriented Software Development community. The main goal of C-SAW is to maintain consistency of complex evolving models. C-SAW offers the ability to explore numerous modelling scenarios by considering crosscutting modelling concerns as aspects that can be rapidly inserted and removed from a model. This permits a modeller to make changes more easily to the base model without manually visiting multiple locations in the model.

ISBN 978-82-14-04396-9

Comparing C-SAW to our approach, there are two differences. C-SAW focuses on specifying *side-effects* rules to deal with model changes, while our approach is focused on capturing various links among different model elements. In addition, C-SAW does not consider explicitly inter-metamodel references.

6.3 XWeave

XWeave is a model weaver based on EMF's ECore meta metamodel [7]. XWeave takes a base model as well as one or more aspect models as input and weaves the content of the aspect models into the base model. XWeave finds pointcuts either by name matching or by defining them with the oAW expression language. Similarly to C-SAW, XWeave uses an automated approach to identifying links between elements of different models while our approach is focused on manual, rigorous trace link establishment.

7 Conclusions & Future Work

In this paper we have presented an approach to establishing trace links between models expressed using different modelling languages. We have proposed using case-specific traceability metamodels that define strongly typed and semantically rich traceability links and inter-model constraints to ensure the soundness of the established links.

We have used the proposed technique for defining rigorous traceability links between requirements specifications expressed using models conforming to the non-trivial i* [18] and KAOS [5] metamodels. Throughout this work we have identified a number of interesting reoccurring patterns in case-specific traceability metamodels. In the future we plan to encapsulate these patterns in a dedicated language that enables specifying traceability metamodels with rigorously-defined semantics at a higher level of abstraction.

References

1. Future Research Topics Discussion. Traceability Workshop, EC-MDA, November 2005. http://www.sintef.no/upload/10558/Future-Research-Topics.pdf.
2. N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafini. Model Traceability. *IBM Systems Journal*, 2006.
3. B Vanhooff, S Van Baelen, W Joosen, Y Berbers. Traceability as Input for Model Transformations. In *Proc. Traceability Workshop, European Conference in Model Driven Architecture (EC-MDA)*, 2007.
4. Christian Nentwich, Licia Capra, Wolfgang Emmerich and Anthony Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151–185, May 2002.
5. R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering. *Proc. ICSE'98 - 20th International Conference on Software Engineering, Kyoto*, 1998.

ISBN 978-82-14-04396-9

6. Eclipse Foundation. Eclipse Modelling Framework Project (emf). http://www.eclipse.org/modeling/emf.
7. I. Groher and M. Voelter. Models and Aspects in Concert. *In Proc. of the 10th Workshop on Aspect-Oriented Modeling co-located with the 6th International Conference on Aspect-Oriented Software Development (AOSD'07), Vancouver, Canada, ACM Press*, 2007.
8. W. Heaven and A. Finkelstein. A UML Profile to Support Requirements Engineering with KAOS. *IEEE Proceedings: Software*, 2004.
9. S. Kent. Model Driven Engineering. *In Michael J. Butler, Luigia Petre, and Kaisa Sere, editors, Proc. of Third International Conference on Integrated Formal Methods (IFM 2002), volume 2335 of LNCS, pages , Springer.*, 2002.
10. D. S. Kolovos, R. F. Paige, and F. Polack. On Demand Merging of Traceability Links wiht Models. *In ECMDA - TW: Traceability Workshop, at European Conference on Model Driven Architecture, Bilbao, Spain*, 2006.
11. D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Eclipse Development Tools for Epsilon. *Eclipse Summit Europe, Eclipse Modeling Symposium*, 20062.
12. A. E. Limon and J. Garbajosa. The Need for a Unifying Traceability Scheme. *In Proc. Traceability Workshop, European Conference in Model Driven Architecture (EC-MDA)*, 2005.
13. D. Del Fabro Marcos, J. Bézivin, F. Jouault, E. Breton, E., and G. Gueltas G. AMW: a Generic Model Weaver. *In Proc. of the 1éres Journées sur l'Ingénierie Dirigée par les Modéles*, 2005.
14. L. Naslavsky, T. A. Alspaugh, D. J. Richardson, and H. Ziv. Using Scenarios to Support Traceability. *In TEFSE '05: Proc. of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering, New York, NY, USA, 2005, ACM Press*, 2005.
15. OMG. Metaobject Facility. http://www.omg.org/mof.
16. IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology. IEEE , New York, 1990.
17. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison Wesley, 1999.
18. E. Yu. Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. *Proc. RE-97 - 3rd International Symposium on Requirements Engineering, Annapolis*, 1997.
19. J. Zhang, Y. Lin, and J. Gray. Generic and Domain-Specific Model Refactoring Using a Model Transformation Engine. *In Model-Driven Software Development - Research and Practice in Software Engineering. Springer Verlag*, 2005.

ISBN 978-82-14-04396-9

Fine Grained Traceability for an MDE Approach of Embedded System Conception

Flori Glitia, Anne Etien and Cédric Dumoulin

LIFL - University of Sciences and Technologies of Lille, France
INRIA Lille - Nord Europe, France

Abstract. MDE approaches are wide-spreading more and more. They allow the generation of code from high level abstraction models, using intermediary models. A development system based on an MDE approach often involves several model transformations composing one or more transformation chains. In order to deal with the complexity in such a system, it is necessary to introduce a traceability mechanism helping both the users and the developers keeping track of elements. Several traceability mechanisms and semantics have already been defined, but they are not appropriate to trace fine grained elements, like properties. In this paper, we have adapted existing traceability mechanisms in order to manage a richer semantics for traceability. We have implemented a generic and partially reusable traceability solution tested in Gaspard transformation chains.

1 Introduction

MDE offers the basic principles and a methodology to use models as main artifacts in the life cycle of complex software systems.

We use MDE to implement Gaspard, a co-design environment for Embedded Systems. In this environment, the hardware architecture and the application are modeled separately at a high level of abstraction using UML2. Next, models are transformed into domain specific models and finally into code that will be used by simulators or generators. Figure 1 is a simplified view of our co-design environment. In the real process several intermediate models representing different levels of abstraction are used, and different kinds of code are targeted. Automatic transformations are involved to go from one abstraction level to another or code. The whole transformation process from the high level model in UML to the generated code constitutes what we call a transformation chain.

The transformations that are implemented in the Gaspard environment are often very complex [9]. We have several input data sprayed into different related classes producing several output data also sprayed into different related classes. Furthermore, we have what we call *Black-Boxes*, i.e. external piece of code, to do complex computations, for example compute repetition factors from matrices, which are difficult to be implemented in a transformation language.

As it is defined in the literature, *traceability is the ability to establish degrees of relationship between two or more products of a development process,*

ISBN 978-82-14-04396-9

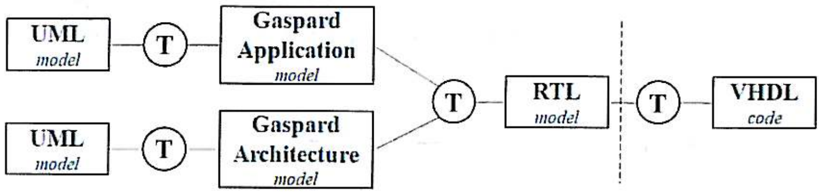


Fig. 1. Simplified View of A Gaspard Transformations Chain

especially products having a predecessor-successor or master-subordinate relationship to one another [10]. Traceability may be used for different purposes such as understanding, capturing, tracking and verification on software artifacts during the development life cycle [1].

In the Gaspard environment, we want to use the trace information for the following purposes:

- *To debug modeled applications* - The traceability links help to establish where a property value or a class instance comes from. Furthermore if a property value is changed in an abstraction level, these links allow users to inspect what has to be modified in other levels.
- *Optionally to debug transformation rules during the rules development process* - Trace can help to know which rules are involved in producing a property value or a class instance.
- *To explore architecture for the evaluation of alternative software/hardware solutions.* - Several architectural solutions as well as several application specifications have to be evaluated with regard to their performance and cost. The evaluation is done at code level but modifications are done at model levels. A traceability mechanism can help in order to identify the property values from the models associated with the values at code level.

As a consequence of these requirements for traceability in our system, our trace solution should provide the possibility to trace not only transformed classes, but also transformed property values. In order to do this we need a fine grained trace mechanism. Two types of transformation are involved in our environment, model to model and model to code. We want to be able to trace both kind, but as our work is in progress, this article will describe a traceability solution only for the model to model transformation.

The traceability solutions proposed in literature do not respond entirely to our needs of fine grained tracing of models elements and transformation debug. We proposed to enhance a *trace metamodel for elements* with new concepts and use a *Global Trace metamodel*[5] for the navigation between models and their traces in a transformation chain. To generate and exploit the trace, we have instrumented our transformation engine and developed an independent visual tool. This tool is organized as Eclipse plug-ins using EMF. It allows selecting a

ISBN 978-82-14-04396-9

class or a property from a model and shows its associated elements along the transformation chain. The models and the tool are independent of the transformation engine and of the way traces are collected. We presume that they are reusable in other projects, as long as the appropriate information for the trace model is provided.

This paper relates our experience in implementing the trace mechanism. It is organized as follows. Section 2 presents existent traceability solutions adaptable to our environment. Section 3 presents the *Local Trace* metamodel and the *Global Trace* metamodel along with the reasons of the metamodels choice. Section 4 presents how we exploit the models in order to recover information and Section 5 presents the future work and concludes.

2 Related Work

Many solutions for traceability are proposed in the literature [1] [3], each of them responding to specific needs of projects.

MDE has as main principle that *everything is a model*, so it seems a good solution to store trace information as models. Solutions are proposed to keep the trace information in the initials models source or target [11]. The major drawbacks of this solution are that it pollutes the models with additional information and it requires adaptation of the metamodels in order to take into account traceability. Using a separate trace model with a specific semantics has the advantage of keeping trace information independent of initial models. [4].

In [3] the authors argue that an optimal solution for a trace metamodel should be a simple core that offers a predefined link between the elements that allows customization and extensibility to define new links. A traceability metamodel should be able to express the links between all the elements in a transformation. The metamodel proposed in [4] can trace model elements but it is not possible to specify traces for the properties of elements.

One solution for collecting the trace information is during the transformations since this only incurs a small cost [7] as the trace model is viewed as an additional target model. As for this reason, trace generation could be manually implemented in transformations to produce an additional trace target model or it can be supported by the transformation engine [2]. In [4], an automatic generation of trace code into rule code is presented, based on the fact that transformation programs are models that could be transformed into another models that contains trace code.

After the trace is generated the main interest for the user is to have simple and quick access to the information that he needs. To interrogate a transformation chain, the trace models, which are produced during the transformations, are not enough. This kind of traceability is referred as *traceability in the small* [5]. For an end-to-end solution for traceability, another type of model, called megamodel, is introduced to handle the navigation between initial models and trace models, referred as *traceability in the large* [5]. In other solutions is proposed to use a trace model for the entire transformation chain, like in [8]. They introduce new

ISBN 978-82-14-04396-9

concepts to manage the navigation in a transformation chain like the abstraction level at which the model elements are found.

A metamodel, UTR (Unified Transformation Representation), is proposed to handle a transformations chain [7]. The main purpose for this metamodel is to facilitate transparent composition and reuse of transformations written in various transformation languages. UTR metamodel could be easily extended with concepts for global traceability [7].

In order to be as little intrusive as possible, we have chosen to have separated trace models for which we have defined appropriate metamodels.

3 Traceability Metamodels Description

In order to implement traceability in a transformation chain, we use the idea of separation of traceability levels between *traceability in the small* and *traceability in the large* [5], which we refer as local and global traceability. We modify the metamodel developed in [4] in order to take into account the fine-grained traces and the black-boxes, and obtain a Local Trace metamodel. For the global navigation we use a Global Trace metamodel.

3.1 Local Trace Metamodel

The Local Trace metamodel capture the traces between the input and the output of one transformation. The metamodel is based on the Trace metamodel presented in [4]. Figure 2 shows the Local Trace metamodel. It contains core concepts that are based on the Trace metamodel and some additional business concepts.

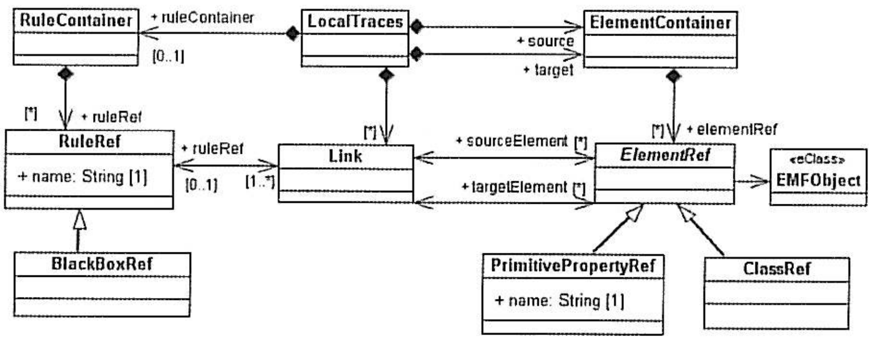


Fig. 2. Local Trace Metamodel

Core concepts. The Local Trace metamodel contains two main concepts *Link* and *ElementRef* in order to express that one or more source elements are

ISBN 978-82-14-04396-9

linked to one or more target elements. Those concepts are the same as in [4]. In our metamodel *ElementRef* is an abstract class representing element that can be traced: instances of Classes or Property values. Property values are traced using *PrimitivePropertyRef* which points to the property container instance and has a name that is the feature name. In fact, only property values that are primitive types in the Java sense, like integer, double, String etc., need special handling because there are no instances in the model. Properties that are typed by a normal class are traced by the *ClassRef* mechanism. Tracing property values is one of the requirements for our trace solution.

Additional business concepts. More information is needed in order to trace the transformation rules and black-boxes. The rule producing the link is traced using *RuleRef* concept. A rule can be associated to several links, so the association many to one between *RuleRef* and *Link*. The *RuleRef* concept is optional and *RuleRef* instances are generated only if they are used. Those kind of elements are generated only for transformation debug.

Black-Boxes are special kind of rules: they have input elements from the input model and produce output elements of the output model. So, they can be traced with *Links*, but we take care to identify the black-boxes with the *BlackBoxRef* concept which is a subclass of *RuleRef*.

RuleRef and *BlackBoxRef* are concepts that allow establishing a bridge between the traceability and the transformation world. Information concerning the rules and black-boxes are used for the transformation debug as an answer to the second purpose of our trace solution presented in introduction.

Implementations concepts. An *ElementRef* has a reference to the real object in the source or target models. As in our environment models are implemented with EMF, the reference named *EMFObject* is an EObject from Ecore metamodel. The *LocalTraces* concept represent the root of the Local Trace model and it contains *RuleContainer*, used as a container for the rules, and two *ElementContainers*, used for containing respectively source and target *ElementRefs*. Separating source and target elements allows reducing the cost of search of input or output elements.

3.2 Global Trace Metamodel

The Global Trace metamodel links Local Traces according to the transformation chain. A Global Trace model is a main entry point from which all the Local Trace models are found, and describes that a target/source model of one transformation is a source/target model for the next/previous transformation.

Figure 3 shows the Global Trace metamodel. It allows to represent all local traces and models in a transformation chain and the way those models are linked, represented by *TraceModel*, respectively *LocalModel*. The models can be shared between transformations, indicating that they are produced by one transformation and consumed by another one.

The *GlobalTraces* implementation concept represents the root of the model.

Introducing this extra Global Trace level [5] permits the navigation between transformed models and their Local Trace models. It also gives a better separa-

ISBN 978-82-14-04396-9

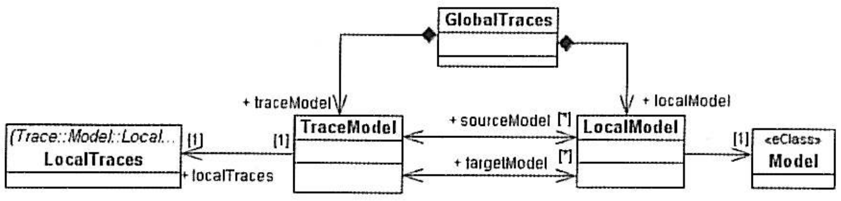


Fig. 3. Global Trace Metamodel

tion of trace information, which leads to a better flexibility for trace creation and exploitation. Not using this Global Trace semantic could have as a consequence collapsing all traces in a big unique trace model for the entire transformation chain, more difficult to create and interrogate.

3.3 Semantics of Traceability Links

The proposed Local Trace metamodel can lead to different Local Trace models for the same transformation, depending on how traces are collected and how rules are organized. More precisely, the difference is in the semantic of the *Link* concept.

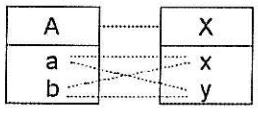


Fig. 4. Simple transformation A to X

In the example in Figure 4, A is transformed into X, x is computed from a and b, and y is also computed from a and b. Such an example can lead to at least three different traces, like in Figure 5, where each tuple *(link; sourceElements; targetElements)* represents a link, its source and target elements.

trace 1	**trace 2**	**trace 3**
– (l1; A; X)	– (l1; A; X)	– (l1; A; X)
– (l2; a,b; x)	– (l2; a; x, y)	– (l2; a; x)
– (l3; a,b; y)	– (l3; b; x, y)	– (l3; b; x)
		– (l4; a; y)
		– (l5; b; y)

Fig. 5. Different set of traces for the same transformation

ISBN 978-82-14-04396-9

The first two traces use multiple sources or targets for the link. In the first trace, link *l2* means that *a* and *b* produces *x*. In the second trace, link *l2* means that *a* produces *x* and *y*. In the third trace, there is a link for each simple relation.

Our metamodel support all of these traces, and the exploitation of any of them leads to the same result. Searching the ancestor of *y* in any set of traces will always have the same result: *a* and *b*. At this point we argue that the way the trace is constructed do not influence the way it is exploited and this give flexibility trace generation.

4 Trace Exploitation

One of the main goals of collecting traces in our system is to permit the user to inspect these traces. After the trace is collected in trace models, the trace models are interrogated using an search algorithm in order to obtain the elements related to an element specified by the user, then the result is shown to the user.

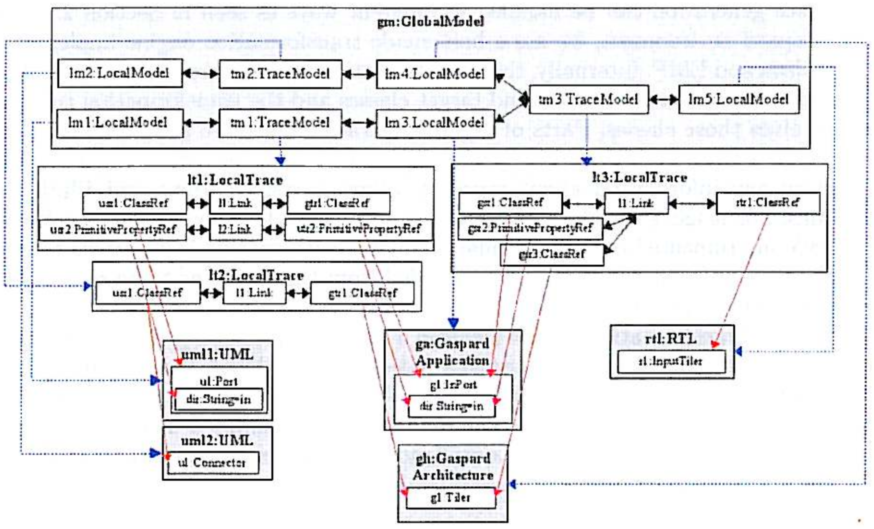


Fig. 6. Schematic view of Local and Global Trace models in a simplified Gaspard transformation chain

In Figure 6 are described the Local and Global Trace models that are produced from the Gaspard transformation chain presented in Figure 1 with the trace tool. The navigation sense between the models is represented with dotted lines.

Due to these relations, starting from the Global Trace model we can navigate to Local Trace models and to the models involved in the transformations. Also in

ISBN 978-82-14-04396-9

a Local Trace model we navigate between models elements. A Local Trace model only contains references to the actual elements in the models, as for example *usr1* is a reference for the element *u1* in model *uml1:UML*. We represent in this example links between the elements without instances of *RuleRef* in order to not overload the figure. One example of link is *(l2, usr2, gtr2)* that is an element of *lt1* model, instance of *LocalTrace*. Links can be constructed between instances of *PrimitivePropertyRef* (for example *usr2* and *gtr2* of *lt1*), instances of *ClassRef* (for example *gsr1* and *rtr1* of *lt3*) or between instances of *PrimitivePropertyRef* and *ClassRef*(for example *gsr3* and *rtr1* of *lt3*).

Adding another model in the transformation chain requires the generation of the trace for this new transformation and the addition of some new information to the Global Trace model. The proposed mechanism allows having a scalable trace adapted to our environment that already contains five transformation chains each made of two to four transformations.

4.1 Trace Generation

Trace generation can be handled in different ways as seen in Section 2. In the Gaspard environment, we use a homemade transformation engine implemented in Java and EMF. Internally, this transformation engine stores information about each transformation: source and target classes and the transformation rule that involves those classes. Parts of the Local Trace models are generated using the internal trace data provided by the engine. Our transformation engine doesn't store any information about property values transformations and Black Box calls, nor it use a standardize way to compute such transformation or calls. We have instrumented our transformation engine with specific instructions to store the trace information that is not provided from the engine as to be recovered in Local Trace models.

Our transformation engine includes a mechanism allowing defining transformation chains from several transformations. The Global Trace model can be generated using this mechanism. The Global Trace model can also be generated from the Local Trace models. The models can be identified using the information from the *ElementsRef* of Local Trace and their order using the fact that an element is *sourceElement* or *targetElement*.

4.2 Search Algorithm

The search algorithm start with the selected element and a search direction that can be *backward* or *forward*. We have a backward navigation in case the navigation is made from the current element to the elements at a higher level of abstraction in the transformation chain (as for example in Figure 1 we go from an element in RTL model to an element in UML model) and forward if we descend in the abstraction level (from an element in UML model to an element in RTL). In the following steps we consider a backward navigation:

1. the algorithm identifies the model containing the element

ISBN 978-82-14-04396-9

2. it searches in the Global Trace model for the *LocalTrace* that has the model as target
3. from this *LocalTrace*, it looks in the *targetContainer* for the *ElementRef* corresponding to the element
4. from the *ElementRef*, it navigates to the *Link*, and then to the source *ElementRefs*
5. for each *ElementRef* found, the algorithm is applied recursively

The recursive call stops when no *LocalTrace* can be found in step 2. In this case, the current element is linked to the first selected element. Also, all intermediate *ElementRef* found are linked to the first selected element. The algorithm is the same for a property value search. In this case, step 3 manipulates a *PrimitivePropertyRef* corresponding to the selected property.

Let us apply this algorithm on the transformation chain described in Figure 6. If *g1:InPort* is the selected element, step 1 identifies the *ga:GaspardApplication* model. Step 2 results in the identification of *tm1:TraceModel*. The *u1:Port* is found during Step 3 and 4. The algorithm is recursively called with the *u1:Port* element but stops as *uml1:UML* is not the target of any *LocalTrace* model. The element that is linked with *g1:InPort* is *u1:Port*.

4.3 Visual Tool

To ease the exploration of the trace, we developed a visual tool in Eclipse. This tool allows to select elements (classes or properties), to run the algorithm and then to presents the search result. The models involved in the transformation are presented as EMF Trees. The property view has been modified in order to be able to select property values. The user selects one or more classes or properties and then presses the search button. The algorithm is run and the result elements or properties are selected in the EMF Trees. The result for the transformation debug is shown in the console, allowing the discovery of the rules involved in the transformation of selected elements. The tool is independent of the way the traces are collected, and we think it can be reused in other projects as long as the trace models respect the specified trace metamodels. The trace models and mechanism are completely transparent to the users.

5 Conclusion and Future Works

In this paper we present our experience in the implementation of traceability for an MDE approach of embedded systems. We reuse ideas presented in literature for the construction of Local and Global Trace models to which we add new concepts like *RuleRef*, *BlackBoxRef* to trace rules and black-box and *PrimitivePropertyRef* to trace property values. The former are used to debug transformation rules whereas the latter is needed for architecture exploration and to debug modeled applications.

We explain why in a complex system, with complex transformations, a fine grained traceability is needed and why the trace are at elements property level.

ISBN 978-82-14-04396-9

Then we present examples and argue our choice of models concepts. We present the search algorithm used to inspect the trace for an element along the transformation chain. We argue that the use of a trace system is helpful as a debug tool for the transformation chains developers as well as for the final application designers and give a clearer image of the relation between the input models and the output models.

We intend to use the trace information for more complex operations as to help us do semi-automatic architecture exploration: Some properties, like number, type, deployment of processors, memory, buses, etc., of the high level application models are tuned according to the result, like speed, power consumption etc., of the simulation of the low level models or code. The trace mechanism will serve to track-back which properties should be changed, and the impact of the changes will help to set the property values.

We are currently in the process of changing our transformation language. As the trace generation is independent from the trace exploitation, the developed algorithm and tool will not affected, as long as the required Local and Global traces are generated. If the new transformation engine provides a trace mechanism we can implement an API to transfer this information to the Local Trace. If it provides its own trace model, a model to model transformation should permit the transfer to our Local Trace Model.

We also investigate how to implement the trace in model to code generation. We think that we will be able to use the same or a very similar Trace metamodels. Indeed, for model to code traceability tracing the classes and property values is mandatory. In addition it will be possible to introduce other concepts required by model to code transformation, like line number.

References

1. Ismenia Galvao, Arda Goknil: Survey of traceability approaches in Model-Driven Engineering, IEEE (2007)
2. K. Czarnecki, S. Helsen: Feature-Based survey of model transformation approaches, IBM System Journal Vol 45, No 3 (2006)
3. N. Aizenbud-Reshef, B.T. Nolan, J. Rubin, Y. Shaham-Gafni: Model traceability, IBM System Journal Vol 45, No 3 (2006)
4. Jouault, Frederic: Loosely Coupled Traceability for ATL, In: Proceedings of the European Conference on MDA Traceability Workshop, Nurnberg, Germany (2005)
5. Mikael Barbero, Marcos Del Fabro, Jean Bezivin: Traceability and Provenance Issues in Global Model Management, In: Proceedings of the European Conference on MDA Traceability Workshop, Haifa, Israel (2007)
6. Vanhooff, B., Van Baelen, S., Joosen, W., Berbers, Y.: T raceability as input for model transformations. In: Proceedings of the European Conference on MDA Traceability Workshop, Haifa Israel. (2007)
7. B. Vanhooff, D. Ayed, S. Van Baelen, W. Joosen, and Y. Berbers, UniTI: A Unified Transformation Infrastructure. In: G. Engels, B. Opdyke, D.C. Schmidt, and F. Weil, ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007), Nashville, TN, USA, 2007.

ISBN 978-82-14-04396-9

8. Jean-Remi Falleri, Marianne Huchard, Clementine Nebut: Towards a traceability framework for model transformations in Kermeta In: Proceedings of the European Conference on MDA Traceability Workshop, Bilbao, Spain (2006)
9. Sebastien Le Beux: Un flot de conception pour applications de traitement du signal systématique implementées sur FPGA á base d'Ingénierie Dirigée par les Modèles, PhD Thesis, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, France, December (2007)
10. A. Geraci: IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries, Institute of Electrical and Electronics Engineers Inc., The, (1991)
11. Y. Velegrakis, R. J. Miller, J. Mylopoulos: Representing and Quering Data transformations, International conference on Data Engineering (ICDE), pp 81-92, April (2005)

ISBN 978-82-14-04396-9

ISBN 978-82-14-04396-9

Defining a Traceability Link Semantics for Design Decision Support

Robert Brcina and Matthias Riebisch

Technical University of Ilmenau, Germany
robert.brcina|matthias.riebisch@tu-ilmenau.de

Abstract. The development and the evolution of large, complex software systems bear several risks. Traceability links can help to master the complexity of these tasks. Currently, they are not used in a large scale, because tool support is necessary to reduce the overhead effort. At present, tools for handling traceability links cannot be effectively developed, because the syntax and the semantics of the traceability links are not sufficiently defined. In this paper we present a set of traceability link types together with a definition of their semantics. The set of link types was developed by analyzing the link evaluation and exploitation. The presented link types are customized for the support of architectural design decisions in regard to a set of non-functional design goals. The extension of the results to a wider scope is discussed. The work was performed within a large industrial project.

1 Introduction

Large, business critical software systems have to perform a tough succession of changes in order to maintain their value for a company. Changes of complex software systems bear several risks. Design decisions have to be made under uncertain conditions, because the consequences of different alternatives cannot be determined precisely. Traceability can support the decision making by facilitating the software comprehension, the change impact analysis, and the minimization of risks. However, the accuracy of the traceability links constitutes a critical issue to achieve the expected benefit. If the information provided by these links is wrong, bad decisions and the introduction of errors are a consequence.

For maintaining the accuracy of the traceability links, tool support during the establishment, the adaptation and the evaluation of the links is necessary. Furthermore, the link maintenance by humans requires a high effort and introduces new risks of mistakes. Currently, an effective tool support can hardly be provided because the semantics of the artefacts and models used, and of the traceability links themselves is not defined precisely enough. Guidelines and rules for link modifications during changes and for evaluations are only provided as far as the semantics of the links and of the linked artefacts are defined. Furthermore, there is a broad variety of proposals for link types, but little attention has been paid on the definition of the link semantics. The semantics has to support the link utilization in order to be of practical value. A pure categorization of links

ISBN 978-82-14-04396-9

constitutes an important step, but it does not lead to the required preconditions for a tool support.

In this paper we present a traceability link definition framework for supporting architectural design decisions. Traceability links have to represent relations between artefacts in different phases of the development process. Depending on the goal of the decision, different types of artefacts and relations have to be considered. Due to space limitations, we focus on architectural design decisions regarding the non-functional design goal evolvability. We start from a subset of the currently used link types and define their semantics by the way of their utilization. Even if the results of this work are applicable only to these decisions, we expect that the discussion about link types and semantics is driven forward and that they lead to a significant support for the tool development.

After a brief discussion of related work, section 3 introduces the traceability approach. Based on them, in section 3.3 the indicators and the corresponding resolution actions are introduced. In section 3.4 the application of the approach in an iterative development process is illustrated by an example. In section 4 we introduce the link meta model and the link semantics.

2 Related Work and Traceability Link Utilization

From the engineering point of view *traceability links* are used to trace design decisions during the development process. Both, *functional* and *non functional tracing* allow following functional and non functional issues of the system development [10]. They facilitate system comprehension by providing the required information about relations between artefacts and entities, e.g. the scenario based approach described in [7]. A *traceability model* is used to define the required entities and relations during the software development, e.g. in [8]. The definition of relations as traceability link types is important for the utilization of the model information. Unfortunately, the definition of a standard set of traceability link types is still an unresolved issue. However, for a tool support of design evolution a semantic differentiation of the traceability link types is needed. Due to different research goals, a high number of traceability link type definitions has been established, e.g. in [9] or [8]. As a step toward simplification and abstraction, we will later restrict ourselves mostly on the *implementedBy* traceability link type plus the dependency relations of the modelling language UML2.

Traceability Link Utilization. Link types should be defined in a way suitable for the intended usage. In the following we list a set of activities in which the links are used, together with the goals of using them.

Verification of (forward) engineering activities: identification of the input to an engineering activity (e.g. requirements, goals, models, risks); understanding and making a decision; verification of the completeness of an activity; verification of the design rules applied.

Change impact analysis: identification of all entities depending on the changed one; understanding the type of dependency to a related entity in order to identify the necessary way of changing it, accordingly.

ISBN 978-82-14-04396-9

Software comprehension and reverse engineering: identification of all related entities to the one in focus; understanding the type of relation between the entity in focus and a related one; identification of abstractions, e.g. design patterns, architectural styles, principles.

Identification of the source of a decision or requirement: identification of the stakeholder who demanded a particular property; justification of a decision effort; resolution of a set of contradicting requirements.

Decision support: understanding the influence factors and the goals of a decision; establishment of proposals for solutions; evaluation of alternative solutions.

System configuration and versioning: identification of constraints between components; identification of necessary changes to resolve a constraint; identification of differences between two versions of the same artefact and their impact on other ones.

3 Traceability Approach for Design Decision Support

3.1 Architectural Design Decisions for Evolvability

Decision-making and assessment are both critical activities for the success of development processes because they apply the success criteria. They are performed during and after work on artefacts. Assessments have to be performed as early as possible to provide early feedback for developers and to minimize rework. They provide the means to control iterative development processes.

Elaboration of the criteria for the assessments. In the following, we will apply evolvability as one criteria for architectural quality in long-term development projects. It is influenced by an appropriate use of the concepts of abstraction, delegation, modularization [4], conceptual integrity [5] and separation of concerns [6]. Beside these aspects, there are additional ones related to general issues of software development processes, such as the availability of a proper set of documentation. We will focus on the concepts of modularization, encapsulation and separation of concerns for the assessment of the goal evolvability. Additionally, the ease of change at architectural level is an important criterion. Problems arise from effects called scattering, tangling and insulated artefacts, which hamper the above mentioned criteria and the quality attributes. Based on a traceability approach and on the set of defined traceability links we introduce indicators (see section 3.3), which enable us to analyze, reveal and reduce these effects.

Artefact Categories. In our approach, we consider relations between requirements, architectural elements and implementation. The key idea is to enable a tracing of all software elements back to the requirements. According to the method used for the system development, different artefacts are involved. Within a feature driven development we consider four types of artefacts: features (F), architectural components (A), classes (C) and implementation artefacts (I). Examples for I type artefacts are configuration units.

Typical effects. From a point of view of changeability, a 1:1 relation between dependent objects is the most effective one (Fig. 1 left). In such a case the change

ISBN 978-82-14-04396-9

of a feature f_0 requires only the change of the precisely related component a_0. Following all related objects, the ideal but usually not realistic case is that each object does not have more than two traceability link connections. In this case the alignment of components to features is possible. It enables minimal invasive changes, as the features can be exchanged by code composition.

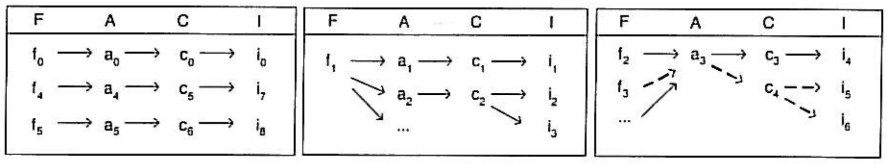


Fig. 1. Ideal case (left), feature scattering (center), feature tangling (right).

In practice, more dependencies have to be considered. A high number of dependencies means that more artefacts are affected by a change resulting in a higher maintenance effort and a reduction of the variability. The two important types of effects are discussed using Fig. 1. Feature scattering (Fig. 1 center) means that one feature f_1 is implemented by more than one architectural component – in this case the components a_1, a_2 and others. In the case of feature tangling, an architectural component is responsible for more than one feature (Fig. 1 right) e.g. the implementation of feature f_2, f_3 and others is tangled in component a_3. If one of these features has to be removed, the component a_3 has to be analyzed and split into appropriate parts, with a much higher effort than just the removal of one component. In order to improve the flexibility of a system concerning feature variability, all variability points should be aligned in a way that each of them is related to exactly one optional feature.

Corresponding artefacts depending on the evolution of one feature, e.g., system components and classes are related by traceability links. The right part of Fig. 1 shows an example: the class traceability enables a discrimination between the relation of c_3 and f_2 (indicated by a solid arrow) from the one between c_4 and f_3 (dashed arrow). An impact analysis for feature-related changes is facilitated by this discrimination.

3.2 Types of traceability links and artefacts

In the following, we will show a cutout from an industrial IT infrastructure project for illustration purposes. For the chosen cutout it is not necessary to consider implementation artefacts. The set of features F and the considered subset $\mathcal{F} \subseteq F$ both are contained in the feature model. The set of architectural software components A and the considered subset $\mathcal{A} \subseteq A$ are part of the architecture model. The set of classes C and the considered subset $\mathcal{C} \subseteq C$ are part of the realization (design) model. The traceability links, which are required for the later evaluation by the indicators (section 3.3) are defined in Table 1.

ISBN 978-82-14-04396-9

Traceability Category	Traceability-Link-Type	Link-Source	Link-Destination	Link-Symbol
Component Traceability	*implementedBy*	Feature	Component	$f \rightsquigarrow \mathcal{A}$
Class Traceability	*implementedBy*	Feature	Classes	$f \rightsquigarrow \mathcal{C}$
Component Require Relation	*use*	Component	Component	$a \mapsto \mathcal{A}$

Table 1. The used set of traceability links.

Component Traceability. Each component contributes to a set of requirements. Such a relationship is expressed by the *implementedBy* traceability link pointing to components that implement a set of features.

Class Traceability. Software components consist of a set of classes and vice versa a class c is related to exactly one software component in order to implement at least one part of a feature. For classes the same traceability link type *implementedBy* is used as for features and components.

Component Require Relation. Similar to class traceability this kind of use of traceability links describes the relationship between two components in which one component needs the others to implement the related feature.

3.3 Metrics for evaluation: Scattering and Tangling

The above defined traceability links are used to establish indicators – often called metrics – which enable us to evaluate architectural design decisions regarding the quality attributes for evolvability. The indicators are accompanied by actions for problem resolution and explained in the following, whereas section 3.4 illustrates their application during the evolutionary development. Due to the space limitation, the insulated features effect cannot be discussed here. The traceability link based indicators defined here, together with a variety of other indicators [11] are applied for design decision support and architectural evaluation.

Feature Scattering. Feature scattering affects the evolvability of a system because the change of one feature demands changes of more than one components, thus leading to higher effort and to a higher probability of mistakes than in the ideal case. On the architectural level, feature scattering refers to a relation between one feature and more than one components. In order to avoid a division by zero while calculating the feature scattering indicator, all insulated features and insulated components have to be removed before. The traceability link type necessary for the indicator is indicated within the definition using the traceability link symbol defined in Table 1.

Definition: Feature Scattering Indicator
$fsca$ is based on $a \in A$, $f \in F$ and is defined as follows:

$$sca\,(f) := |\{a : f \rightsquigarrow a\}| - 1, \text{ and} \tag{1}$$

ISBN 978-82-14-04396-9

$$fsca\,(F) := \frac{\sum\limits_{f \in F} sca\,(f)}{|F| \cdot |A|}, \; fsca \in [0,1)\,. \qquad (2)$$

The more features are scattered into components, the worse the evolvability of the software gets and the closer the result of the indicator moves to 1. The maximum value of 1 is reached if $|a|$ approaches infinity and each feature $f \in F$ is implemented by all $a \in A$.

Resolution: Reducing feature scattering

A reduction of the feature scattering could be achieved (a) by splitting up the features into several ones starting with the feature with the highest value for $sca\,(f)$, (b) by merging components to reduce the number of involved components.

Feature Tangling. Feature tangling refers to relations between more than one feature and one component. In order to avoid a division by zero while calculating the feature tangling indicator, all insulated features and insulated components have to be removed before.

Definition: Feature Tangling Indicator
The Feature Tangling Indicator $ftang$ is defined as follows for $a \in A$:

$$tang\,(a) := |\{f \in F : f \rightsquigarrow a\}| - 1, \text{ and} \qquad (3)$$

$$ftang\,(A) := \frac{\sum\limits_{a \in A} tang\,(a)}{|F| \cdot |A|}, \; ftang \in [0,1)\,. \qquad (4)$$

The more features are tangled to one component, the more difficult is the adaptation of this component and the closer is the result of the indicator to one. The maximum value of 1 would be reached if $|f|$ would approach infinity and each component $a \in A$ would implement all features $f \in F$. An example for $ftang$ is shown in section 3.4.

Resolution: Reducing feature tangling

A reduction to the feature tangling effect could be achieved by (i) splitting up the component starting with the highest number of $tang\,(a)$ into several components, each with a reference to the corresponding feature, or (ii) by merging features to reduce the number of involved features.

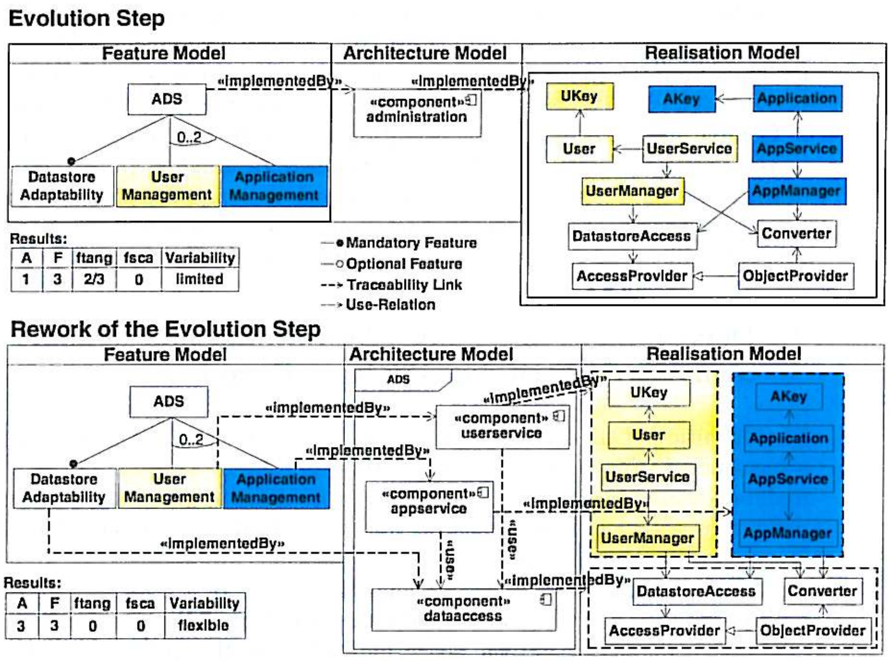
3.4 Illustrating Example

We illustrate the application of our traceability approach by evaluating the architectural solution of an Administration System (ADS) part of an IT-Infrastructure by using the indicators defined in the previous section. We will examine one development iteration (we call it evolution step) and its rework. The goals are evolvability and the ease of change on architectural level in order to support

ISBN 978-82-14-04396-9

variability. Due to space limitations only a part of the IT infrastructure project is visible in the example.

Evolution Step. All three features *User* and *Application Management* and *Datastore Adaptability* are implemented at the levels of features, architecture and classes. The feature *Application Management* allows to manage all application specific information, whereas the feature *User Management* allows to manage all user specific information of the IT infrastructure. As shown in Fig. 2, traceability links are used to express the dependencies between these models.



Results:

A	F	ftang	fsca	Variability
1	3	2/3	0	limited

Results:

A	F	ftang	fsca	Variability
3	3	0	0	flexible

Fig. 2. The evolution step (above) and its rework (below).

Change operations within each evolution step are recorded by traceability links. A feature tangling effect is revealed by three traceability links starting from each feature to the component *administration*. As defined in Table 1 there are two types of traceability links, *use* between components and *implementedBy* links between features and components as well as classes. Additionally, relations between classes are considered.

The evaluation results using the indicators are summarized in Fig. 2 in small tables at the lower left. The result $ftang = 2/3$ indicates that there is a tangling between the features and the architectural component, which hampers the evolvability of the ADS system. In addition to evolvability it is important to achieve a high flexibility regarding changes of features. To support variability

ISBN 978-82-14-04396-9

at architectural level, the components shall be used independently; and the customers can select a random set of optional features, e.g. *User Management* or *Application Management* and their combination. The application of the indicators reveals that the system has a limited variability because of the feature tangling: the two features *User* and *ApplicationManagement* have to be deployed even if only one is needed. In this case the component *administration* consists of classes from both features, as indicated in Fig. 2 by corresponding shades of grey. With the resolution actions these limitations will be resolved by the following rework.
Activities for improvement. The tangling has to be eliminated by splitting the component *administration* into three architectural components. The traceability links indicate that a decomposition into components related to features is possible. The result of the resolution actions is presented in the lower left of Fig. 2. Comparing the results of the evolution step with those of the rework shown in the tables, we state a resolution of the feature tangling from $ftang = 2/3$ down to $ftang = 0$. As a result of the rework it is possible to use all optional features without an overhead effort for configuration. However, the success with this variability goal causes a dependency between the components.

4 Development of a Link Meta Model and a Link Semantics for Design Decision Purposes

The definition of the traceability link semantics - together with the definition of a metamodel - has a big influence on the resulting overhead effort for the link maintenance and management. Following the goals of effort minimization, the definitions are as lightweight as possible. This leads to link semantics with as few as possible, but as many as necessary aspects covered.

We have discussed only a very small subset out of the bandwidth of traceability link utilizations as mentioned in section 3. However, we are able to give application details within the space limitations of this paper. Even if the established link metamodel will have a limited scope, our procedure provides an example for the development of traceability link frameworks based on the intended ways of utilization.

For the design decision support regarding evolvability we only need links of the types *implementedBy* and *use.* As explained in section 3, only links between artefacts of the type feature, component and class are evaluated for a calculation of the indicators. Information about the source and the destination of each link is sufficient for this purpose. For the establishment and the evaluation of different alternatives for design decisions, additional traceability links are required to evaluate the priority of competing design goals. These links connect components and classes to features and are of the type *implementedBy.* As additional information, these links carry design decisions about the way in which influence factors are considered. During the mentioned activities for improvement, new traceability links are established however they do not cover more information or additional types than already mentioned.

ISBN 978-82-14-04396-9

The necessary information for the traceability link can be represented in a metamodel definition. Due to the limited scope we expect less information than mentioned in works with a broader scope, e.g. our earlier work [2]. The resulting definition is expressed using UML as a metamodel as shown in Fig. 3.

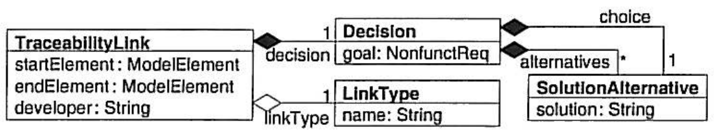


Fig. 3. Traceability Link Metamodel.

The semantic information is covered by the definitions of the link types and artefact types, as well as by the rules which apply to the links. We can distinguish rules at different levels: regarding context-free syntax, context-sensitive syntax, static and dynamic semantics. Rules regarding dynamic semantics are not considered so far. As examples, we mention a few rules related to syntax:

1. Each feature is related to one or more components by an *implementedBy* link.
2. Each component is related to one or more classes by an *implementedBy* link.
3. Each component is related to one or more features by an *implementedBy* link.
4. Each class is related to one or more components by an *implementedBy* link.

The rules for link semantics express design rules. We can distinguish rules of different categories: (a) Very general rules representing general engineering principles, e.g. decomposition and abstraction, (b) method-specific rules, e.g. a rule about the mapping of non-functional goals to functional solution principles according the architectural method by Bosch [1], and (c) domain-specific rules, e.g. a rule which type of response is valid for a certain type of event in a specific telecommunication protocol. We have to state, that only a few rules have criteria which are precise enough to enable clear statements. They can be implemented in tools for an automatic evaluation and verification of models. Most of the precise rules belong to the syntax-related ones. Unfortunately most semantic rules cover less strict criteria; therefore they provide only degrees of fulfillment between true and false. They can be used for human inspection only, but they provide valuable hints and enable a reduction of the search space. Therefore they increase the efficiency of an inspection by reducing the effort and by enabling a concentration and an increased precision.

We just want to mention that there is another type of rules frequently called heuristics which have even less clear criteria. They are used for other ways of link utilization e.g. for impact analysis which are out of the scope of this paper. They are applied e.g. for controlling how far links are tracked and to what level of detail links are maintained, depending on an actual risk.

ISBN 978-82-14-04396-9

5 Conclusion and Future Work

In this paper we have shown the application of traceability links for the support of architectural design decisions. Evolvability was considered as an example of a quality property of architectures. By an example it was shown how links have to be defined in terms of syntax and semantics, to provide the best support for architectural decisions. Even if the purpose is in some way specific, the procedure can be expanded to further ways of link utilization.

The idea of model-based development behind our research aims the coverage of all necessary information in models and the used traceability links. One could dispute that this leads to heavy-weight development processes with a high modeling effort, but a strong restriction to the necessary parts of information and an exhaustive utilization of the models helps to increase the overall efficiency. The next steps of our work include the investigation of the necessary adoptions for other ways of link utilization including a refinement and a revision of the defined link semantics, the evaluation of applicability of these link definitions for the other utilization activities, and the extension of the rule set during empirical research. The link definitions represent a prerequisite to the development of a comprehensive tool support to provide a (partly) automated link management.

References

1. Bosch, J.: Design and Use of Software Architectures – Adopting and evolving a product-line approach. Addison-Wesley(2000)
2. Maeder, P., Philippow, I., Riebisch M.: Customizing Traceability Links for the Unified Process. QoSA 2007, USA, 2007. Springer LNCS, pp. 47-64 (2007)
3. Riebisch, M.: Supporting Evolutionary Development by Feature Models and Traceability. Proc. ECBS2004, IEEE Computer Soc., pp. 370-377 (2004)
4. Parnas, D. L.: On the criteria to be used in decomposing systems into modules. Commun. ACM 15, 12, 1053-1058 (1972)
5. Brooks, F. P.: The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley (1995)
6. Ossher, H., Tarr, P.: Multi-Dimensional Separation of Concerns and the Hyperspace Approach. Commun. ACM vol. 44, 10, 43-50 (2001)
7. Egyed, A.: A Scenario-Driven Approach to Traceability. ICSE'01, pp. 123-132 (2001)
8. Ramesh, B., Jarke, M.: Toward Reference Models for Requirements Traceability. IEEE Trans. Softw. Eng. vol. 27, 1, pp. 58-93 (2001)
9. Klaus Pohl: PRO-ART: Enabling Requirements Pre-Traceability. ICRE'96, IEEE Computer Soc., pp. 76-84 (1996)
10. Pinheiro, F. A. C.: Requirements traceability. In Requirements traceability in Perspectives on Software Requirements. Julio C. S. P. Leite and Jorge Doorn, Kluwer Academic Publishers, pp 91-113 (2004)
11. Matthias Riebisch and Robert Brcina: Optimizing Design for Variability Using Traceability Links. ECBS, IEEE Computer Society, pp. 235-244 (2008)

ISBN 978-82-14-04396-9

Building Model-Driven Engineering Traceability Classifications

Richard F. Paige, Gøran K. Olsen, Dimitrios S. Kolovos, Steffen Zschaler and Christopher Power

Department of Computer Science, University of York, UK.
{paige, dkolovos, cpower}@cs.york.ac.uk
SINTEF, Oslo, Norway. Goran.K.Olsen@sintef.no
TU Dresden, D-01062 Dresden, Germany. szschaler@acm.org

Abstract. Model-Driven Engineering involves the application of many different model management operations, some automated, some manual. For developers to stay in control of their models and codebase, trace information must be maintained by all model management operations. This leads to a large number of trace links, which themselves need to be managed, queried, and evaluated. Classifications of traceability and trace links are an essential capability required for understanding and managing trace links. We present a process for building traceability classifications for a variety of widely used and accepted operations (both automated and manual) and show the results of applying the process to a rich traceability context.

1. Introduction

Traceability is the ability to chronologically interrelate uniquely identifiable entities in a way that matters. The IEEE Standard Glossary of Software Engineering Terminology [IEEE 2004] defines traceability as *"the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match."* Thus, traceability refers to the capability for tracing artefacts along a set of chained operations, where these operations may be performed manually (e.g., crafting a software design for a set of software requirements) or with automated assistance (e.g., generating code from a set of abstract descriptions). In the context of Model Driven Engineering (MDE), many of the artefacts of interest are *models*, conforming to a metamodel, and are constructed using a set of modelling tools. Traceability in MDE is therefore predominantly concerned with chronologically defined relationships involving models and elements of models. The relationships between models are often called *trace links* [Olsen 2007]. However, when applying MDE, we start development of models from other kinds of artefacts: informal, natural language descriptions of requirements, spreadsheets, etc. Traceability needs to consider these artefacts as well, in terms of how models can be traced to other (non-model) artefacts and how (non-model) artefacts can be traced to models.

ISBN 978-82-14-04396-9

Generating and maintaining traceability information is important in order to help control the wealth of different artefacts in the development process: as systems become more complex, and as the application of MDE techniques within a process becomes more in depth, the need for better management of MDE artefacts increases. Traceability helps us to understand the many *dependencies* that exist between MDE artefacts. If we are able to support *end-to-end* traceability—that is, between all artefacts developed and generated in an MDE systems development process—then we can support a variety of different kinds of analysis; for example, showing that a requirement is fulfilled in implementation or showing that artefacts are up to date.

In a realistic MDE context, it is likely that a large amount of traceability information will be generated or created; understanding and managing this information will therefore be challenging, and will require structure to be imposed in order to understand the most appropriate ways to manage it. Traceability information can be better understood and managed through the help of a *traceability classification.* Several classifications have been published (e.g., for requirements engineering) [Ramesh 2001, Limon 2005, Walderhaug 2006]; these vary from abstract, conceptual classifications that help to systematise our understanding of the traceability problem domain, to concrete classifications (or traceability *metamodels*) that help to manage trace information in an implementation. Our focus in this paper is on the *process of building* traceability classifications, and on using this to classify both manual and automated MDE operations, thus helping to enable the full vision of end-to-end traceability.

The structure of the rest of the paper is as follows. In Section 2 we briefly review related work. In Section 3 we sketch a simple process for building traceability classifications. In Section 4 we outline how we have applied the process to build a traceability classification for the MODELPLEX[1] project, encompassing both manual and automated trace links.

2. Related Work

Different styles of traceability classifications have been presented in the literature. In particular, classifications given in terms of *scenarios of use* of traceability are postulated by [Olsen 2007, Walderhaug 2006]. Classifications in terms of specific *domains* have been produced by [Ramesh 2001] for requirements engineering, and for business applications [Rummler 2007].

Traceability classifications in MDE have been developed that emphasise different *attributes* or *characteristics* of traceability. In particular, two categories of classifications can be identified in the literature: classifications that focus on *explicit trace links* (which are captured directly in models themselves using a suitable concrete syntax), and *implicit trace links* where trace information is generated or arises due to application of one or more model management operations. The classification we build in Section 4 includes both explicit and implicit trace links.

More generally, traceability has been identified as an important research issue. The European project AMPLE, focusing on Aspect-Oriented and Model-Driven product

[1] http://www.modelplex.org/

ISBN 978-82-14-04396-9

line engineering aims to support traceability across software product lines [Rummler 2007]. The Grand Challenges in Traceability report [GCT 2006] identifies a number of challenges for managing and maintaining trace information, including evolution of trace information, trace link semantics, and eliciting trace knowledge.

Many trace tools have been developed for managing trace information. Some of the most well-known and widely used include Reqtify [ChiasTek 2007], RequisitePro [IBM 2007], and Acceleo Pro [Obeo 2007]. An approach to trace link generation has been presented by Egyed [Egyed 2003], who presents a trace tool that automatically derives traces from code through requirements. [Kolovos 2006] presents support for trace links where trace information is stored separate from the model; [Jouault 2005] outlines a loosely coupled trace scheme for model transformations.

3. A Traceability Classification Process

As suggested by the related work presented in Section 2, a number of traceability classifications have been presented, but there is little guidance yet on how to systematically build and maintain them (ranging from conceptual models to concrete metamodels). A demonstrated *process* for building traceability classifications is useful for this, not only for building classifications in the first place, but for maintaining classifications as new MDE operations, new relationships between MDE artefacts, and new stakeholder requirements, arise. In order to support this, we first describe a very simple process for building and maintaining traceability classifications, and then in the next section we use it to develop a classification for the stakeholder requirements of MODELPLEX.

The simple process is called the *Traceability Elicitation and Analysis Process (TEAP)*. It is derived from a process developed in [Chan 2005] for elicitation and understanding different forms of model-based contracts. The aim of TEAP is to elicit and analyse traceability relationships in order to determine how they fit into a traceability classification. While eliciting new traceability relationships, we improve our understanding of the *key attributes* of these traceability relationships: the artefacts they involve, their semantics, and their domain of applicability.

When applying TEAP, we typically bootstrap from a simple traceability classification or metamodel, and iteratively and incrementally refine the classification through a number of TEAP *cycles*. Each cycle in the TEAP enriches the existing classification in terms of one or more key attributes of interest.

TEAP is a *triggered* process, in which there are three main activities in each iteration: Elicitation, Analysis, and Classification. In elicitation, we identify basic trace links and relationships. In the second phase, we extrapolate from these the key characteristics of traceability (based on our current understanding) and as a result of this analysis, identify constraints on relationships and any generalisations of relationships and trace links. In the third phase, we build a classification. From this, we can iteratively enrich, refactor, and improve the classification, for example when customer requirements dictate.

TEAP is intended for use in building new classifications *and* for maintaining classifications. The classification we describe for MODELPLEX is generic, and may be

ISBN 978-82-14-04396-9

useful in a variety of settings – however, it can and will be extended over the duration of MODELPLEX, and TEAP can be used to support this.

TEAP is derived from the spiral software development model, due to Boehm, and the spiral model in requirements engineering. The main difference is that TEAP produces metamodels and classifications, rather than software or requirements. The advantage of defining and using TEAP is that it gives extensibility to its product. This is essential in providing a generic, flexible framework for classifying and managing traceability. In other words, the traceability classification can be kept up-to-date by carrying out additional TEAP cycles.

As mentioned earlier, TEAP is a *triggered* process; we can identify when TEAP cycles should be executed. The triggers for executing TEAP cycles include:

- a new model management operation has been defined, in which case cycles should be executed in order to refine the classification
- the system development process has changed; thus cycles should be executed in order to refine how to handle sequences of model management operations.
- one or more modelling languages have changed to include new model relationships, artefacts, or changed model relationships, in which case cycles should be executed to refine and extend the explicit link classification.
- The existing classification does not capture all requirements for traceability inherent in a domain or project context.

The TEAP process is meant to provide guidance, not to dictate the way in which the traceability classification must be extended and refined.

4. Example: Building the MODELPLEX Classification

In this section we outline how we used TEAP (from Section 3) to build a conceptual traceability classification for use in the MODELPLEX European project. We start with a very short overview of the traceability requirements for MODELPLEX, then summarise a few iterations of TEAP applied to these requirements.

4.1 MODELPLEX traceability requirements

MODELPLEX is a three-year integrated project funded by the European Commission, with a mandate to improve productivity in the development of complex systems through use of MDE. The project is case-study driven, with four industrial partners - SAP, Telefonica, Western Geco, and Thales Information Systems - providing real complex system scenarios, to which MDE technology (e.g., architectural modelling, model transformation, performance analysis, simulation, model composition) is to be applied. Each case study has traceability requirements. These can be summarised as:

- the ability to record traceability information that results from applying model management operations: model-to-model (M2M) transformations, model-to-text (M2T) transformations, compositions, simulations, and refinements;
- the ability to manually create trace links between MDE artefacts, e.g., between an architectural model and a use case model, between a weaving model and a design

ISBN 978-82-14-04396-9

model. Manual creation of trace links can involve modelling tools, or the use of a textual domain-specific language tailored for one or more of the case studies;

- the ability to (typically manually) create trace links between MDE artefacts (e.g., models) and non-MDE artefacts (e.g., requirements stored in a MANTIS repository, PDF documents). This is a necessary requirement as some of the MODELPLEX partners do not currently use MDE technologies in their everyday practice; moreover, non-MDE artefacts will always play a substantial role in the MDE process, e.g., for early requirements elicitation and description.
- the ability to store and retrieve trace links and trace metadata from a repository.

We decided to initially produce a conceptual traceability classification that addressed the basic information that needed to be recorded for the first three sets of requirements above. This would then be refined and implemented in a traceability tool, which also provided repository features.

4.2 Basis for TEAP iterations

To apply TEAP, we initially constructed simple traceability infrastructure that would evolve over the TEAP iterations. This infrastructure is depicted in Fig. 1. It is, effectively, a very simple traceability metamodel that expresses the fundamental concepts of *artefacts, trace links,* and *operations.* We specialise this model in the following subsections. *Artefacts* may be both MDE artefacts (e.g., domain-specific models) and non-MDE artefacts (e.g., spreadsheets), and operations (either manual or automated) elaborate the traceability information to be recorded. Finally, the different kinds of trace links will be discussed in the following sections.

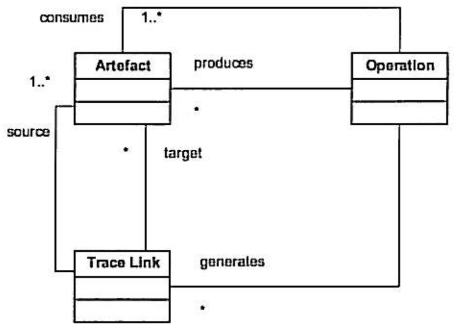


Fig. 1: Basic traceability classification infrastructure

The classification in Fig. 1 is generic, and could thus be used to produce a variety of specialised traceability classifications for different domains and contexts. In each case, TEAP can be used to maintain and extend the basic infrastructure for new domain-specific requirements and contexts.

ISBN 978-82-14-04396-9

4.3 Adding explicit and implicit traceability relationships

We start to extend our simple traceability infrastructure by carrying out TEAP cycles. Our initial cycle was triggered by the obvious observation that the trace link model in Fig. 1 did not satisfy all MODELPLEX requirements for traceability. We thus carried out elicitation (what general kinds of trace links exist?), analysis (what information did these trace links require?), and built a simple classification. The cycle focused on the notions of implicit and explicit traceability. Implicit traceability involves trace links that are created and manipulated by application of MDE operations. Explicit traceability is defined in terms of trace links that are concretely represented in models. Therefore, our initial TEAP cycle was very simple and refines the traceability infrastructure to that shown in Fig. 2.

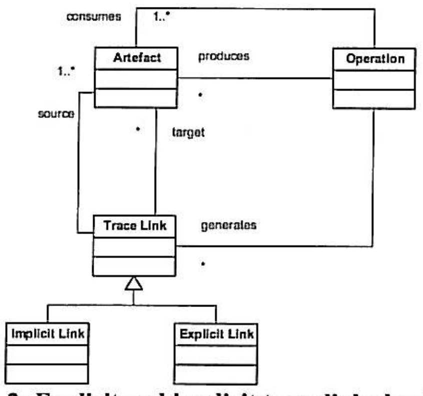


Fig. 2: Explicit and implicit trace link classification

4.4 Iterations for implicit trace link classification

The next cycle was triggered by two observations: the classification of implicit trace links was weak; and, by obtaining more precise requirements about the operations that were to be supported in MODELPLEX. We thus carried out a TEAP cycle for improving our classification of implicit trace links. As usual, there is elicitation (what kinds of MDE operations are relevant?), analysis (what information do operations require, and how should this information be constrained?), and classification.

MDE operations implicitly define a variety of different trace links between two or more artefacts (note that many artefacts will be models, e.g., for model-to-model transformation, but non-model artefacts such as code and requirements may be involved too). An MDE operation takes a set of artefacts as input (if the artefacts are models, they may be from one or more different modelling languages) and produces a set of artefacts and a set of trace links as output. Trace links can be recorded either in the source or target artefacts, or as a separate model [Kolovos 2006]. The basic MDE operations are elicited from studying the relevant standards—particularly UML, MOF, and QVT—which indicate how trace links can be generated. The operations we initially identify are: *query*, *transformation*, *composition* (sometimes called *merging*),

ISBN 978-82-14-04396-9

update (also called *update-in-place*), *creation*, *deletion*, *model-to-text*, and *sequences* of operations. The resulting classification (focusing strictly on subclasses of Implicit Link from Fig. 2) is shown in Fig. 3. As well, new operations (subclasses of Operation) are added for each, e.g., Query Operation, Delete Operation, etc.

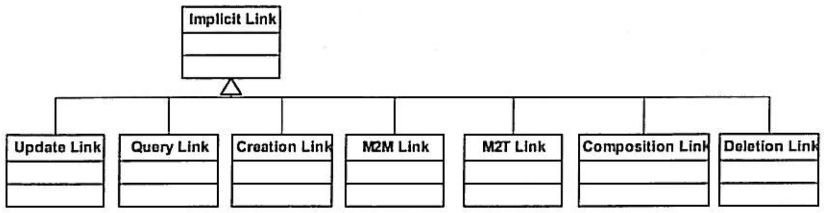


Fig. 3: Additions for implicit trace links

Examples of the well-formedness constraints elicited and produced in the analysis phase are shown below.

```
context QueryOperation inv:
   self.consumes->forAll(a |
      self.generates->exists(t | t.source->includes(a) and
      self.produces->includesAll( t.target )));
   self.generates->forAll( t | t.oclIsTypeOf(QueryLink))

context CreationOperation inv:
   self.consumes->isEmpty();
   self.generates.target->includesAll(self.produces);
   self.generates->forAll( t | t.oclIsTypeOf(CreationLink));
```

4.4 Iterations for explicit traceability relationships

We next carried out a TEAP cycle for improving our classification of explicit trace links. This cycle was triggered by refined MODELPLEX requirements for explicit representation of trace information in models and in domain-specific languages. Recall that explicit trace links are explicitly defined between artefacts, using one or more languages. For example, a UML dependency constitutes a specific kind of explicit trace link. Obviously, there are many different kinds of explicit trace links, and many of them will be domain specific (and language specific). We illustrate the results of the TEAP process for MODELPLEX's explicit trace links. As was the case for implicit trace links, we carry out elicitation (what kinds of explicit trace links are relevant?), analysis (what information do these links require?), and classification.

The initial elicitation and analysis identified two basic kinds of explicit trace links: *model-model* links (e.g., the aforementioned UML dependency), and *model-artefact* links (e.g., between a model and a spreadsheet). Trace links entirely between non-model artefacts were determined to be out of scope, and managed by other tools.

The model-model links were then further analysed. These were determined to be divisible into *static* links (which represent structural relationships that do not change over time) and *dynamic* links, which represent information regarding models that may

ISBN 978-82-14-04396-9

change over time. A variety of both static and dynamic links were collected from MODELPLEX's requirements. Some examples of static model-model links are:

- *consistent-with* links, where two models must remain consistent with each other, e.g., a sequence and class diagram.
- *dependency* links, where the structure and meaning of one model depends on a second. Dependency links can be further subdivided into: *is-a* links (e.g., subtyping), *has-a* links (e.g., references), *part-of* links, *import* and *export* links, *usage* links (e.g., one component uses another's services), *refinement* links (e.g., where a component reduces non-determinism in a second component).

Some examples of dynamic model-model links include: *calls* links (where one model calls behaviours provided by a second model), *notifies* links (where it is necessary to record information that cannot be handled automatically, such as changes that require human intervention). Furthermore, there are design-time relationships, such as *generates* or *builds* links that indicate where information from one model is used to produce or deduce the second model; and *synchronized-with* relationships, where behaviours between models are synchronized. These usually apply when there is some kind of tracking mechanism introduced between models. A further example includes the *consistent-with* trace links that can exist between an early requirements specification such as those in i*, and models of functional requirements [Alencar, 2000].

Model-artefact links are important in MODELPLEX, to support trace links between MDE artefacts (including UML models as well as domain-specific models) and non-MDE artefacts, particularly spreadsheets, requirements databases, and results of simulations. The scope of model-artefact links is broad, and we did not attempt to elicit all such links in our classification. We provide important links in MODELPLEX, while giving a classification that can be extended in the project.

The intent of most model-artefact links is to enable coverage checking, e.g., of requirements. This is the case in MODELPLEX. The trace links of interest in MODELPLEX were the following:

- *satisfies* links, to indicate that properties or requirements captured in an artefact are satisfied by a model. Variants on satisfies links include *verifies* links (which involve a specific mechanism, such as testing) and *certifies* links (which also link to external standards and arguments for safety or security).
- *allocated-to* links, used when information in a non-model artefact is allocated to a specific model that represents the information.
- *performs* links, indicating that a task described in an artefact is carried out by a specified model
- *explains* and *supports* links, indicating that, e.g., a model is explained by a non-model artefact (e.g., natural language documentation).

These trace links are summarised in Fig. 4 (focusing strictly on the explicit trace link part of the classification).

ISBN 978-82-14-04396-9

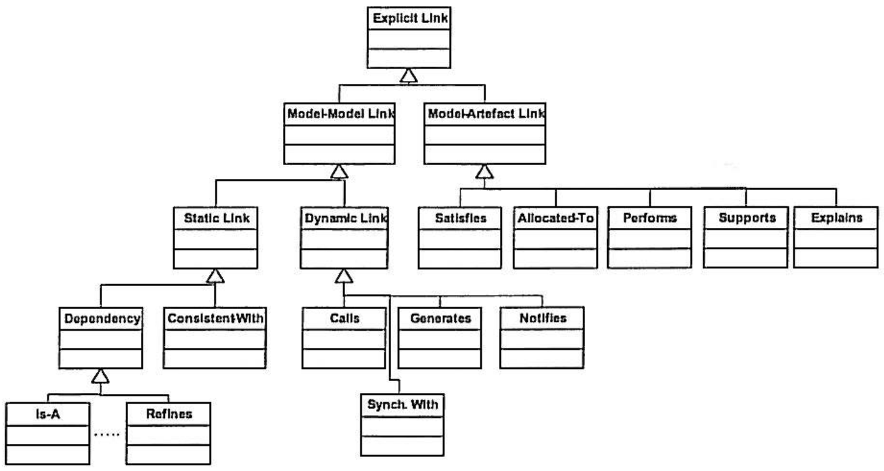


Fig. 4: Summary of explicit trace links

As this brief discussion suggests, the space of explicit trace links is rich and complicated, and encompasses many domain- and language-specific characteristics.

5. Conclusions

We have presented a lightweight process for building traceability classifications, and illustrated its application to a conceptual classification for the MODELPLEX process. The classification identifies basic categories of traceability – implicit and explicit – and populates these categories with trace links from different MDE operations (such as transformation and query) and from different modelling scenarios relevant to MODELPLEX. The classification developed above is a living document, and will be extended iteratively and incrementally over the course of the project. Furthermore, it will be *refined* from a conceptual classification to a concrete design that can be supported in a trace tool that includes a repository and capabilities for retrieving, storing, and updating trace links.

TEAP has so far proved to be suitably lightweight, yet helpful in guiding the construction and the iterative improvement of traceability classifications. Since most, if not all, classifications must evolve with time, the value of an iterative and incremental process for evolving classifications is substantial.

Acknowledgments

This research has been co-funded by the European Commission within the 6th Framework Programme project MODELPLEX contract number 034081.

References

[Alencar, 2000] F. Alencar, J. Castro, G. Cysneiros, J. Mylopoulos. From Early Requirements Modeled by the i* Technique to Later Requirements Modeled in Precise UML. In *Proc. of Workshop de Engenharia de Requisitos 1ed. Brasil,* Brazil, 2000.

[Aizenbud-Reshef 2005] N. Aizenbud-Reshef, R.F. Paige, J. Rubin, Y. Shaham-Gafni, and D.S. Kolovos. Operational semantics for traceability. In *Proc Workshop on Traceability 2005*, Nuremberg, Germany, November 2005.

[Aizenbud-Reshef 2006] N. Aizenbud-Reshef, B. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Systems Journal* 45(3), 2006: pp 515-526.

[Borland 2007] CaliberRM: http://www.borland.com.

[Chan 2005] Z.E. Chan and R.F. Paige. Designing a Domain-Specific Contract Language: a Metamodelling Approach, in *Proc. EC-MDA 2005*, LNCS 3748, Springer-Verlag, November 2005.

[Chiastek 2007] Chiastek, Reqtify: http://www.chiastek.com/products/reqtify.html

[Egyed 2003] A. Egyed, A Scenario-Driven Approach to Trace Dependency Analysis. *IEEE Transactions on Software Engineering*, 2003. 29: p. 17.

[GCT 2006] Grand Challenges in Traceability, Center of Excellence for Traceability http://www.traceabilitycenter.org/downloads/documents/GrandChallenges/

[IEEE 2004] IEEE Std 610.12-1990 IEEE Standard Glossary of Software Engineering Terminology, 2004.

[Jouault 2005] F. Jouault. Loosely Coupled Traceability for ATL. In *Proc. ECMDA-FA Workshop on Traceability*, SINTEF Technical Report STF90, November 2005.

[Kolovos 2006c] D.S. Kolovos, R.F. Paige, and F.A.C. Polack On-Demand Merging of Traceability Links with Models. *ECMDA 06 Traceability Workshop* Bilbao, 2006.

[Limon 2005] A. Limon and J. Garbajosa. The Need for a Unifying Traceability Scheme. In *Proc. Workshop on Traceability 2005*, Nuremberg, November 2005.

[Obeo 2007] Acceleo Pro Traceability, http://www.acceleo.org

[Olsen 2007] G. Olsen and J. Oldevik. Scenarios of traceability in model to text transformations. In *Proc. ECMDA-FA 2007*, LNCS, Springer-Verlag, June 2007.

[Walderhaug 2006] S. Walderhaug, U. Johansen, E. Stav, and J. Aagedal. Towards a generic solution for traceability in MDD. In *Proc. Workshop on Traceability 2006*, Bilbao, Spain, July 2006.

ISBN 978-82-14-04396-9

Change Impact Analysis based on Formalization of Trace Relations for Requirements

Arda Goknil, Ivan Kurtev, Klaas van den Berg

Software Engineering Group, University of Twente, 7500 AE Enschede, the Netherlands
{a.goknil, kurtev, k.g.van.den.berg}@ewi.utwente.nl

Abstract. Evolving customer needs is one of the driving factors in software development. There is a need to analyze the impact of requirement changes in order to determine possible conflicts and design alternatives influenced by these changes. The analysis of the impact of requirement changes on related requirements can be based on requirements traceability. In this paper, we propose a requirements metamodel with well defined types of requirements relations. This metamodel represents the common concepts extracted from some prevalent requirements engineering approaches. The requirements relations in the metamodel are used to trace related requirements for change impact analysis. We formalize the relations. Based on this formalization, we define change impact rules for requirements. As a case study, we apply these rules to changes in the requirements specification for Course Management System.

Keywords: change impact analysis, requirements traceability, requirements metamodels

1 Introduction

Change management is a prerequisite for high-quality software development. Changes may be caused by changing user requirements and business goals or be induced by changes in implementation technologies. There is a need to analyze the impact of requirement changes in order to determine possible conflicts and design alternatives influenced by these changes.

The analysis of the impact of requirement changes on other requirements can be based on requirements traceability. Requirements relations can be used as trace links to determine the impact of requirements change. Current trace metamodels and mechanisms consider relations between model elements mostly without assigning any semantics. The lack of semantics in trace links causes imprecise results in change impact analysis and explosion of impacts problem [3].

We propose a requirements metamodel with well defined types of requirements relations. This metamodel represents the common concepts extracted from some prevalent requirements engineering approaches. The requirements relations are used to trace related requirements for change impact analysis. We formalize these requirements relations. Based on this formalization, we define change impact rules for

ISBN 978-82-14-04396-9

requirements. We aim at more precise analysis with these rules. As a case study, the rules are applied to changes in the requirements specification for Course Management System.

The paper is structured as follows. Section 2 gives details of the requirements metamodel. Section 3 gives the formalization for the requirements relations and consistency constraints. In Section 4, we describe the change impact rules derived from the formalization of requirements relations. In Section 5, we give a case study to illustrate the change impact analysis. Section 6 presents the related work. Section 7 concludes the paper and describes future work.

2 Requirements Metamodel

The requirements metamodel contains common concepts identified in existing requirements modeling approaches [26] [12] [19] [14] [27]. The metamodel in Fig. 1 includes entities such as *Requirement*, *Stakeholder* and *Relationship* in order to model general characteristics of requirements artifacts.

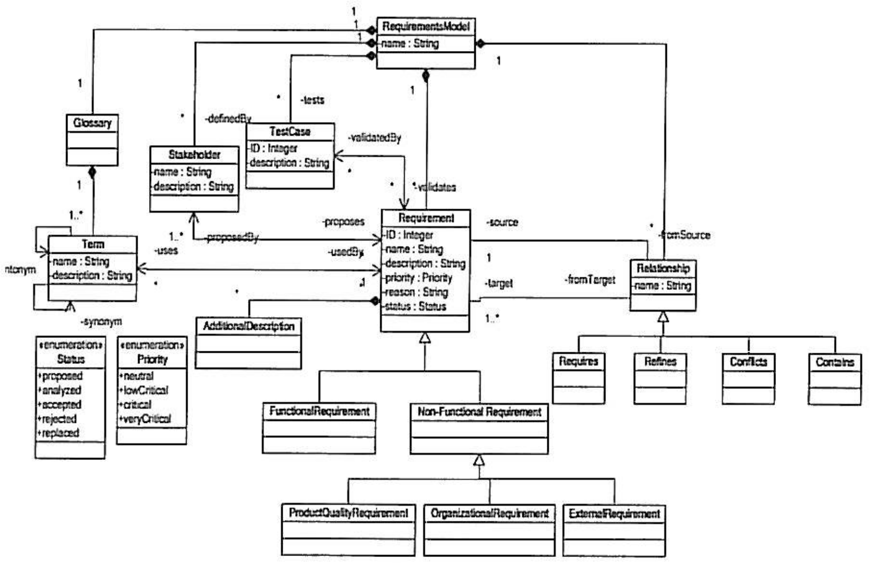


Fig. 1. Requirements Metamodel

In this metamodel, all requirements are captured in a requirements model (*RequirementModel*). A requirements model is characterized by a name property and contains requirements instances of the *Requirement* entity. All requirements have a unique identifier (*ID* property), a name, a textual description (*description* property), a priority, a rationale (*reason* property), and a status. Requirements may have additional descriptions (*AdditionalDescription* entity) such as a use case or any other formalization. Usually, requirements are classified as functional and non-functional

ISBN 978-82-14-04396-9

requirements. Non-functional requirements may come from required characteristics of the software (product quality requirements), the organization developing the software (organizational requirements) or from external sources [22]. Requirements can be related with each other. We recognize four types of relations: *Refines*, *Requires*, *Conflicts*, and *Contains*. These core relations can be specialized and new relations may be added as specializations of the *Relationship* concept. The metamodel includes the entities *Stakeholder*, *TestCase*, *Glossary* and *Term*. Test cases are not always considered as parts of requirements specifications. However, they are important to validate or verify requirements. Some metamodels [19] [27] consider test cases as a part of the requirements specification.

3 Formalization of Requirements Relations

In this section, we give the definitions and formalizations of requirements relations in Fig. 1. These formalizations make it possible to understand various types of dependency between requirements provided by the requirements relations. This understanding helps us to specify more precise change impact rules for requirements. The relations in the requirements metamodel are defined and formalized as follows:

- *Definition 1. Requires relation*: A requirement R_1 *requires* a requirement R_2 if R_1 is fulfilled only when R_2 is fulfilled. R_2 can be treated as a pre-condition for R_1 [27].
- *Definition 2. Refines relation*: A requirement R_1 *refines* a requirement R_2 if R_1 is derived from R_2 by adding more details to it [26].
- *Definition 3. Contains relation*: A requirement R_1 *contains* requirements $R_2..R_n$ if R_1 is the conjunction of the contained requirements $R_2..R_n$. This relation enables a complex requirement to be decomposed into child requirements [19].
- *Definition 4. Conflicts relation*: A requirement R_1 *conflicts with* a requirement R_2 if the fulfillment of R_1 excludes the fulfillment of R_2 and vice versa [25].

The definitions given above are intuitive and informal. In the remaining part of this section we give a formal definition of requirements and relations among them in order to derive sound change impact rules.

We assume the general notion of requirement being "a property which must be exhibited by a system" [8]. We define a requirement R as a tuple <P, S> where P is a predicate (the property) and S is a set of systems that satisfy P, i.e. $\forall s \in S : P(s)$.

- *Formalization of Requires*

Let R_1 and R_2 are requirements such that $R_1 = <P_1, S_1>$ and $R_2 = <P_2, S_2>$. R_1 *requires* R_2 iff for every $s_1 \in S_1$ then $s_1 \in S_2$.

From this definition we conclude that $S_1 \subset S_2$. The subset relation between the systems S_1 and S_2 gives us the properties of *non-reflexive*, *non-symmetric*, and *transitive* for the *requires* relation.

ISBN 978-82-14-04396-9

- *Formalization of Refines*

Let R_1 and R_2 are requirements such that $R_1 = <P_1, S_1>$ and $R_2 = <P_2, S_2>$. We assume that P_1 and P_2 are formulas in first order logic (there may be formalizations of requirements in other types of logics such as modal and deontic logic [18]) and P_2 can be represented in a conjunctive normal form in the following way:

$$P_2 = p_1 \wedge p_2 \wedge \ldots \wedge p_{n-1} \wedge p_n \wedge q_1 \wedge q_2 \wedge \ldots \wedge q_{m-1} \wedge q_m$$

Let $q^1_1, q^1_2, \ldots, q^1_{m-1}, q^1_m$ are the predicates such that $q^1_i \rightarrow q_i$ for $i \in 1..m$

R_1 *refines* R_2 iff P_1 is derived from P_2 by replacing every q_i in P_2 with q^1_i $i \in 1..m$ such that the following two statements hold:

(a) $P_1 = p_1 \wedge p_2 \wedge \ldots \wedge p_{n-1} \wedge p_n \wedge q^1_1 \wedge q^1_2 \wedge \ldots \wedge q^1_{m-1} \wedge q^1_m$

(b) $\exists s \in S_2 : s \notin S_1$

From the definition we conclude that if P_1 holds for a given system s then P_2 also holds for s. Therefore $S_1 \subset S_2$. Similarly to the previous relation we have the properties *non-reflexive, non-symmetric, transitive* for the *refines* relation. Obviously, if R_1 *refines* R_2 then R_1 *requires* R_2.

- *Formalization of Contains*

Let R_1, R_2 and R_3 are requirements such that $R_1 = <P_1, S_1>$, $R_2 = <P_2, S_2>$, and $R_3 = <P_3, S_3>$. We assume that P_2 and P_3 are formulas in first order logic and can be represented in a conjunctive normal form in the following way:

$$P_2 = p_1 \wedge p_2 \wedge \ldots \wedge p_{m-1} \wedge p_m$$
$$P_3 = p_{m+1} \wedge p_{m+2} \wedge \ldots \wedge p_{n-1} \wedge p_n$$

R_1 contains R_2 and R_3 iff P_1 is derived from P_2 and P_3 as follows:

$P_1 = P_2 \wedge P_3 \wedge P'$ where P' denotes properties that are not captured in P_2 and P_3 (i.e. we do not assume completeness of the decomposition [26])

From the definition we conclude that if P_1 holds then P_2 and P_3 also hold. Therefore, $S_1 \subset S_2$ and $S_1 \subset S_3$. Obviously, the *contains* relation is *non-reflexive, non-symmetric*, and *transitive.*

- *Formalization of Conflicts*

Let R_1 and R_2 are requirements such that $R_1 = <P_1, S_1>$ and $R_2 = <P_2, S_2>$. Then, R1 *conflicts with* R2 iff $\neg\exists s : s \in S_1 \wedge s \in S_2 : P_1(s) \wedge P_2(s)$. The *conflicts* relation is *symmetric.*

It should be noted that the definition of *requires* is given in extensional terms as a subset relation between the systems that satisfy the requirements. The definitions of *refines* and *contains* are given in intensional terms, that is, they take into account the form of the requirement specification as a predicate. If we would interpret *refines* in an extensional way then we will conclude that *requires* and *refines* are both interpreted as a subset relation and therefore are equivalent. Apparently in our formalization, *refines* and *requires* are different.

Several constraints for the consistency of relations can be specified based on the formalizations of the relations for the requirements metamodel. These inconsistencies are different from the *conflicts* relation between requirements. Inconsistencies, here,

ISBN 978-82-14-04396-9

indicate that relations between requirements are violating their constraints. Some of the constraints for the consistency of relations and their proofs (proof by contradiction) are given below:

Constraint 1: (R_1 Refines R_2) $\rightarrow \neg$ (R_2 Requires R_1)
Proof: Let R_1 *refines* R_2 and suppose R_2 *requires* R_1. According to the formalization of the refines relation (R_1 Refines R_2) $\rightarrow$ (R_1 Requires R_2). The *requires* relation is non-symmetric. Contradiction.

Constraint 2: (R_1 Requires R_2) $\rightarrow \neg$ (R_1 Conflicts R_2)
Proof: Let R_1 *requires* R_2, then, by the formalization of the *requires* relation $S_1 \subset S_2$. Suppose R_1 *conflicts* R_2, then, by the formalization of the *conflicts* relation $\neg\exists s : s \in S_1 \wedge s \in S_2 : P_1(s) \wedge P_2(s)$. Contradiction.

Constraint 3: (R_1 Requires R_2) $\rightarrow \neg$ (R_2 Contains R_1)
Proof: Let R_1 *requires* R_2, then, by the formalization of the *requires* relation $S_1 \subset S_2$. Suppose R_2 *contains* R_1, then, by the formalization of the *contains* relation $S_2 \subset S_1$. Contradiction.

Constraint 4: (R_1 Refines R_2) $\rightarrow \neg$ (R_1 Contains R_2)
Proof: Let R_1 *refines* R_2. According to the formalization of the *refines* relation, q^1_1, q^1_2, ..., q^1_{m-1}, q^1_m are the predicates such that $q^1_i \rightarrow q_i$ for $i \in 1..m$. Suppose R_1 *contains* R_2, then, by the formalization of the *contains* relation $P_1 = P_2 \wedge P'$. Contradiction.

Constraint 5: (R_1 Refines R_2) $\rightarrow \neg$ (R_2 Contains R_1)
Proof: Let R_1 *refines* R_2. According to the formalization of the *refines* relation, q^1_1, q^1_2, ..., q^1_{m-1}, q^1_m are the predicates such that $q^1_i \rightarrow q_i$ for $i \in 1..m$. Suppose R_2 *contains* R_1, then, by the formalization of the *contains* relation $P_2 = P_1 \wedge P'$. Contradiction.

Constraint 6: (R_1 Refines R_2) $\rightarrow \neg$ (R_1 Conflicts R_2)
Proof: Let R_1 *refines* R_2, then, by the formalization of the refines relation $S_1 \subset S_2$. Suppose R_1 *conflicts* R_2, then, by the formalization of the *conflicts* relation $\neg\exists s : s \in S_1 \wedge s \in S_2 : P_1(s) \wedge P_2(s)$. Contradiction.

Constraint 7: (R_1 Contains R_2) $\wedge$ (R_1 Refines R_3) $\rightarrow \neg$ (R_2 Refines R_3)
Proof: Let R_1 contains R_2 and R_1 refines R_3 then, by the formalization of the *contains* and the *refines* relations:

(a) $P_1 = P_2 \wedge P'$
(b) $P_3 = p_1 \wedge p_2 \wedge ... \wedge p_{n-1} \wedge p_n \wedge q_1 \wedge q_2 \wedge ... \wedge q_{m-1} \wedge q_m$
(c) $P_1 = p_1 \wedge p_2 \wedge ... \wedge p_{n-1} \wedge p_n \wedge q^1_1 \wedge q^1_2 \wedge ... \wedge q^1_{m-1} \wedge q^1_m$ where q^1_1, q^1_2, ..., q^1_{m-1}, q^1_m are the predicates such that $q^1_i \rightarrow q_i$ for $i \in 1..m$.

Suppose R_2 refines R_3, then, by the formalization of the *refines* relation:

(a) $P_3 = p_1 \wedge p_2 \wedge ... \wedge p_{n-1} \wedge p_n \wedge q_1 \wedge q_2 \wedge ... \wedge q_{m-1} \wedge q_m$

ISBN 978-82-14-04396-9

(b) $P_2 = p_1 \wedge p_2 \wedge ... \wedge p_{n-1} \wedge p_n \wedge q^2_1 \wedge q^2_2 \wedge ... \wedge q^2_{m-1} \wedge q^2_m$ where q^2_1, $q^2_2, \ldots, q^2_{m-1}, q^2_m$ are the predicates such that $q^2_i \rightarrow q_i$ for $i \in 1..m$.

Since $P_1 = P_2 \wedge P'$, P_1 includes all predicates in P_2. Contradiction.

These constraints above are derived from the formalization of the relations. There may be other types of constraints which originated from domain. For instance, one product quality requirement requires at least one functional requirement in the requirements model since product quality requirements come from the required characteristics of the software. These kinds of constraints are not explicitly stated in the requirements metamodel in Fig. 1. These can be defined using the OCL (Object Constraint Language) [30]:

```
-- all product quality requirements require at least one functional
-- requirement
context ProductQualityRequirement
inv: fromSource->size()>0 and
     fromSource->forAll(rl|rl.oclIsTypeOf(Requires) and
     rl.target->forAll(rq|rq.oclIsTypeOf(FunctionalRequirement))
```

In [7], we propose a prototyping that can perform reasoning on requirements that may detect implicit relations and inconsistencies on the basis of the formalization of relations and constraints. We also propose an approach for customizing the requirements metamodel in order to support different requirements specifications. Furthermore, our approach for customization keeps the semantics of the core concepts intact and thus allows reuse of tools and reasoning over the customized metamodel. We express the metamodels as OWL [5] ontologies. The composition operator is also expressed in OWL since this language allows direct mapping from set operations to language constructs. By using OWL we can use the reasoning capabilities of the ontology tools. We specified OWL [5] ontologies for each metamodel with Protégé [6] environment. Inference rules were expressed in SWRL [10]. The rules to check the consistency of relations were implemented as SPARQL [24] queries. The inference rules are executed by Jess rule engine [11] available as a plug-in in Protégé. To reason upon the requirements, the user specifies them as individuals (i.e., instances) in ontology. The inference and consistency checking rules are executed on this ontology.

4 Change Impact Analysis based on the Formalization of Relations

In this section, we give change impact rules for requirements based on the formalism of requirements relations. A change introduced to a model element can be in one of two phases [4]: "*A proposed change implies that impact analysis should be performed to determine how change would impact the existing system, whereas an implemented change implies that all impacted artifacts and their related links should be updated to reflect the change*". In this paper, we aim at giving some rules to the requirements engineer about the possible impacts of a proposed requirements change. For the

ISBN 978-82-14-04396-9

change impact analysis we also propose a distinction for impacted elements and candidate impacted elements: "*A candidate impacted element is the element identified as possibly impacted by a proposed change and it should be checked*". Another classification for impacted elements is direct/indirect impact. A direct impact occurs when the model element affected is related by one of the dependencies that fan-in/out directly to/from the changed model element [3]. An indirect impact occurs when the element is related by the set of dependencies representing an acyclic path between the changed and effected elements [3]. This type of impact is also referred as an N-level impact where N is the number of intermediate relationships. We propose step by step process to analysis the impacted elements. Requirements engineer first consider the directly impacted elements by using the impact rules with tool support, then do the changes in the impacted elements if needed. The previous indirectly impacted elements in 2-level impact are directly impacted elements in the second step. N-level impact analysis can be done by processing direct impact N-times in this way. These classifications for impacted elements are orthogonal. Table 1 gives the classification of impacted elements in the context of requirements modeling.

Table 1 Classification of Impacted Elements in the Context of Requirements Modeling

	Directly Impacted Elements	**Indirectly Impacted Elements**
Candidate Impacted Elements	All relations and requirements in 1-level impact should be considered to be updated.	All relations and requirements in N-level (N > 1) impact should be considered to be updated.
Actual Impacted Elements	All relations and requirements in 1-level impact are impacted and they must be updated.	All relations and requirements in N-level (N > 1) impact are impacted and they must be updated.

There are two types of changes in the requirements model:

- Changes in the requirements entities
    - A new requirement is added.
    - A requirement is deleted.
    - A requirement is modified (one or more of the predicates are deleted, or new predicates are added, or both)
- Changes in the requirements relations
    - A new relation is added.
    - A relation is deleted.
    - A relation is modified (type of the relation is changed)

Modifying a requirement is mainly about changing the text of requirement which is depicted by 'description' attribute of the Requirement entity in Fig. 1. There are other attributes of the Requirement entity such as 'name', 'priority', 'status' and 'reason'. However, we do not take into account changes in these attributes of a requirement in the paper. Since the Relation entity has only a type, changes in relations is only about changing types of the relation. Table 2 gives the change impact rules for requirements.

ISBN 978-82-14-04396-9

Table 2. Change Impact Rules for Requirements

		Case 1	Case 2	Case 3	Case 4
		R1 contains R2 and R3	**R1 refines R2**	**R1 requires R2**	**R1 conflicts R2**
sub a.	**R1 is deleted**	R2 and R3 are the candidate impacted requirements. The relation is impacted and it must be deleted.	R2 is not impacted. The refines relation is impacted and it must be deleted.	R2 is not impacted. The relation is impacted and it must be deleted.	R2 is not impacted. The conflicts relation is impacted and it must be deleted.
sub b.	**R2 is deleted**	R1 and R3 are candidate impacted requirements. The relation is impacted and it must be deleted.	R1 is the candidate impacted requirement. The relation is impacted and it must be deleted.	R1 is the impacted requirement. The relation is impacted and it must be deleted.	R1 is not impacted. The conflicts relation is impacted and it must be deleted.
sub c.	**R1 is modified**	R2 and R3 are the candidate impacted requirements. The relation is the candidate impacted relation.	R2 is not impacted. The relation is the candidate impacted relation.	R2 is not impacted. The relation is the candidate impacted relation.	R2 is not impacted. The conflicts relation is the candidate impacted relation.
sub d.	**R2 is modified**	R1 and R3 are the candidate impacted requirements. The relation is the candidate impacted relation.	R1 is the candidate impacted requirement. The relation is the candidate impacted relation.	R1 is the candidate impacted requirement. The relation is the candidate impacted relation.	R1 is not impacted. The relation is the candidate impacted relation.
sub e.	**New R added**	If R is a Product Quality Requirement then R1, R2, and R3 are candidate container requirements for R.	If R is a Product Quality Requirement then R1, R2 and R3 are candidate container requirements for R	If R is a Product Quality Requirement then R1, R2, and R3 are candidate container requirements for R.	If R is a Product Quality Requirement then R1, R2, and R3 are candidate container requirements for R.
sub f.	**Relation between R1 and R2 is deleted**	There is no impacted requirement. The relations inferred from it are the impacted relations.	There is no impacted requirement. The relations inferred from it are the impacted relations.	There is no impacted requirement. The relations inferred from it are the impacted relations.	There is no impacted requirement. The relations inferred from it are the impacted relations.
sub g.	**Relation between R1 and R2 is modified**	There is no impacted requirement. The relations inferred from it are the impacted relations.	There is no impacted requirement. The relations inferred from it are the impacted relations.	There is no impacted requirement. The relations inferred from it are the impacted relations.	There is no impacted requirement. The relations inferred from it are the impacted relations.
sub h.	**New relation is added to R1**	There is no impacted requirement.	There is no impacted requirement.	There is no impacted requirement.	There is no impacted requirement.

ISBN 978-82-14-04396-9

It should be noted that we only consider direct impacts for the change impact rules given in Table 2. We derive the change impact rules given in Table 2 from the formalizations of relations. Due to space limitation, we only explain some of them in the following.

Case 2 sub a: R_1 is deleted while R_1 *refines* R_2
Impact: Let R_1 *refines* R_2, then, by the formalization of the *refines* relation $q^1_1, q^1_2, \ldots, q^1_{m-1}, q^1_m$ are the predicates such that $q^1_i \rightarrow q_i$ for $i \in 1..m$. Deleting R_1 means deleting the predicates of R_1 including $q^1_1, q^1_2, \ldots, q^1_{m-1}, q^1_m$. This change does not imply any impact on the predicates of R_2. We conclude that R_2 is not impacted. Since there should not be any dangling relation in the model, the *refines* relation is impacted and it must be deleted.

Case 2 sub b: R_2 is deleted while R_1 *refines* R_2
Impact: Let R_1 *refines* R_2, then, by the formalization of the *refines* relation $q^1_1, q^1_2, \ldots, q^1_{m-1}, q^1_m$ are the predicates such that $q^1_i \rightarrow q_i$ for $i \in 1..m$. Since R_1 includes more general predicates ($q^1_i \rightarrow q_i$), deleting the predicates of R_1 including p_i and q_i may imply that the predicates (p_i and q^1_i) in R_1 are impacted. We conclude that R_1 is the candidate impacted requirement. Since there should not be any dangling relation in the model, the *refines* relation is impacted and it must be deleted.

Case 3 sub b: R_2 is deleted while R_1 *requires* R_2
Impact: Since the definition of *requires* is given in extensional terms as a subset relation between the systems that satisfy the requirements, for every $s_1 \in S_1$ then $s_1 \in S_2$ when R_1 *requires* R_2. Deleting the predicates of R_2 may imply deleting or changing some of the predicates of R_1. We conclude that R_1 is the candidate impacted requirement. Since there should not be any dangling relation in the model, the *requires* relation is impacted and it must be deleted.

Case 1 sub c: R_1 is modified while R_1 *contains* R_2 and R_3
Impact: Let R_1 *contains* R_2 and R_3, then, by the formalization of the *contains* relation $P_1 = P_2 \wedge P_3 \wedge P'$. Modifying the predicate P_1 affects either the equation or the predicates P_2 or P_3. We conclude that R_2 or R_3 are the candidate impacted requirements and also the *contains* relation is the candidate impacted relation.

Case 2 sub c: R_1 is modified while R_1 *refines* R_2
Impact: Let R_1 *refines* R_2, then, by the formalization of the *refines* relation $q^1_1, q^1_2, \ldots, q^1_{m-1}, q^1_m$ are the predicates such that $q^1_i \rightarrow q_i$ for $i \in 1..m$. Changing the predicates of R_1 (p_i, q^1_i) does not affect the predicates of the more general requirement R_2 but may have an impact on the refines relation. The refines relation may not be valid anymore since the predicates of R_1 and R_2 may not ensure the formalism of the *refines* relation. We conclude that R_2 is not impacted. The refines relation is the impacted relation. It should be noted that we assume if there is a change in the general and refined requirements, requirements engineer always start changing requirements from the general ones.

ISBN 978-82-14-04396-9

Case 3 sub c: R_1 is modified while R_1 *requires* R_2
Impact: Since the definition of *requires* is given in extensional terms as a subset relation between the systems that satisfy the requirements, for every $s_1 \in S_1$ then $s_1 \in S_2$ when R_1 *requires* R_2. According to this subset relation R_1 has no implication on R_2. We conclude that R_2 is not impacted by modifying R_1. Since the subset relation may not be valid anymore, the *requires* relation is the candidate impacted relation.

Case 4 sub d: R_2 is modified while R_1 *conflicts* R_2
Impact: Since the definition of *conflicts* is given in extensional terms as an exclusive disjunction relation between the systems that satisfy the requirements ($\neg\exists s : s \in S_1 \wedge s \in S_2 : P_1(s) \wedge P_2(s)$), the changes in R_2 has no impact on R_1. Since the exclusive disjunction relation may not be valid anymore because of the change, the *conflicts* relation is the candidate impacted relation.

Case 2 sub d: R_2 is modified while R_1 *refines* R_2
Impact: Let R_1 *refines* R_2, then, by the formalization of the *refines* relation q^1_1, q^1_2, ..., q^1_{m-1}, q^1_m are the predicates such that $q^1_i \rightarrow q_i$ for $i \in 1..m$. Modifying the predicates of R_2 (p_i, q_i) may have an impact on the predicates of R_1 (p_i, q^1_i) since there should be implication relation ($q^1_i \rightarrow q_i$) between refined predicates of R_1 & R_2. Or the refines relation may not be valid anymore. We conclude that R_1 is the candidate impacted requirement and *refines* is the candidate impacted relation.

Case 1 sub d: R_2 is modified while R_1 *contains* R_2 and R_3
Impact: Let R_1 *contains* R_2 and R_3, then, by the formalization of the *contains* relation $P_1 = P_2 \wedge P_3 \wedge P'$. Modifying the predicate P_2 affects either the equation or the predicates P_2 or P_3. We conclude that R_2 and R_3 are the candidate impacted requirements and also the *contains* relation is the candidate impacted relation.

5 Case Study Course Management System

In this section we apply the proposed approach in a case study. An existing requirements specification document is represented as a model instance of the requirements metamodel in Section 2. We also compared the benefits of our approach for change impact analysis with the benefits of having limited type of relations provided by some commercial tools such as IBM Rational RequisitePro. RequisitePro provides only two relations between requirements: *traceFrom* and *traceTo*. The relations in our requirements metamodel (e.g., the refines relation) must all be mapped to one of those two relations. The case study is about the requirements for Course Managements System. This system supports the basic facilities such as enrolling for a course, uploading roster and course materials, grading students, sending e-mails to students. The system supports three types of end users: administrator, student and lecturer. Fig. 2 gives the requirements model of the partial requirements specification of the course management system. Requirements for the model can be found in the appendix. Due to the page limitation and to simplify the

ISBN 978-82-14-04396-9

case study, we do not give the whole requirement specification in Fig. 2 and appendix.

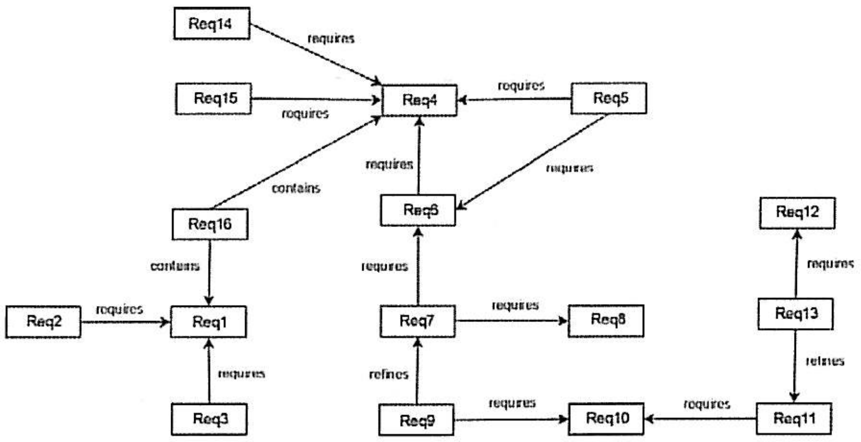


Fig. 2. Partial Requirements Model for the Course Management System Requirements Specification

We have three change scenarios (deleting Req6, modifying Req9 and Req11) in the requirements model given in Fig. 2 for the following requirements:

Req6. The system shall maintain a list of events the students can be notified about.

Req9. The system shall notify students about the events of the lectures they are enrolled for.

Req11. The system shall allow lecturers to send e-mail to students enrolled for the lecture given by that lecturer.

These are some of the change scenarios and impacts of the changes according to the change impact analysis given in Table 2:

Change 1: Deleting Req6
Impact: For the change in Req6, we consider the outgoing and incoming relations (Req6 requires Req4, Req5 requires Req6, Req7 requires Req6) of Req6 and the requirements (Req4, Req5, and Req7) related to Req6. According to the change impact rules given in Table 2, we determine the following impacts:

- *Req6 requires Req4 (Case 3 sub a)*: Since the required requirement is not impacted by deleting the requiring requirement, Req4 is not impacted. There should not be any dangling relation in the model. Therefore, the *requires* relation is impacted and it must be deleted.
- *Req5 requires Req6 (Case 3 sub b)*: Since the requiring requirement is the impacted requirement by deleting the required requirement, Req5 is impacted. It can not be satisfied by the system without specifying and satisfying Req6.
- *Req7 requires Req6 (Case 3 sub b):* Req7 is impacted. The *requires* relation is impacted and it must be deleted.

ISBN 978-82-14-04396-9

The directly impacted elements for *change 1* are Req5 and Req7. The second step is to determine indirectly impacted elements by *change 1*. The indirectly impacted elements in the first-step are directly impacted in this step.

- In the first step, we determine that Req5 is impacted by deleting Req6. When we analyze the impact Req5 can not be satisfied without satisfying Req6. We also decide to delete Req5. According to this change, we have the following impact:
  - *Req5 requires Req4 (Case 3 sub a)*: Req4 is not impacted. The *requires* relation is impacted and it must be deleted.
- We also determine that Req7 is impacted by deleting Req6. Req7 can not be satisfied without satisfying Req6. We also decide to delete Req7. We have the following impact:
  - Req7 *requires Req8 (Case 3 sub a)*: Req8 is not impacted. The *requires* relation is impacted and it must be deleted.
  - *Req9 refines Req7 (Case 2 sub b)*: Req9 is the candidate impacted requirement. The *refines* relation is impacted and it must be deleted.

When we analyze the direct impacts of *change 1* in RequisitePro, we do not have a distinction for types of relations. Therefore, we can not eliminate some of the related requirements to Req6. We identify all requirements (Req4, Req5 and Req7) related to Req6 as impacted requirements.

Change 2: Modifying Req9. We have the following requirement by modifying Req9:
Req9. *The system shall notify students about the events of school activities and lectures they are enrolled for.*
Impact: For the change in Req6, we consider the outgoing and incoming relations (Req9 refines Req7, Req9 requires Req10) of Req9 and the requirements (Req7 and Req10) related to Req9. We determine the following impacts:

- *Req9 refines Req7 (Case 2 sub c)*: Since the refined requirement is not impacted by modifying the refining requirement, Req7 is not impacted. Modifying the predicates of Req9 does not affect the predicates of the more general requirement Req7 but may have an impact on the refines relation. The refines relation may not be valid anymore since the predicates of Req9 and Req7 may not ensure the formalism of the *refines* relation. The refines relation is the candidate impacted relation.
- *Req9 requires Req10 (Case 3 sub c)*: Since the required requirement is not impacted by deleting the requiring requirement, Req10 is not impacted. Since the subset relation between Req9 and Req10 derived from the formalization of *requires* may not be valid anymore, the *requires* relation is the candidate impacted relation.

The candidate directly impacted elements for *change 2* are the *requires* and *refines* relations. The second step is to determine indirectly impacted elements by *change 2*. When we analyze the modification of Req9, we determine that these two relations are not impacted actually. Since we do not have any directly impacted requirements and relations, there are no indirectly impacted requirements and relations.

In RequisitePro, we identify all requirements (Req7 and Req10) related to Req9 as impacted requirements when we analyze the directly impacted elements. We can not identify the candidate impacted relations because we do not have the semantics of relations.

ISBN 978-82-14-04396-9

Change 3: Modifying Req11. We have the following requirement by modifying **Req11**: *The system shall allow lecturers to send e-mail and sms messages to students enrolled for the lecture given by that lecturer.*

Impact: For the change in Req11, we consider the outgoing and incoming relations (Req11 requires Req10, Req13 refines Req11) of Req11 and the requirements (Req10 and Req13) related to Req11. We determine the following impacts:

- *Req11 requires Req10 (Case 3 sub c)*: Since the required requirement is not impacted by deleting the requiring requirement, Req10 is not impacted. Since the subset relation between Req11 and Req10 derived from the formalization of *requires* may not be valid anymore, the *requires* relation is the candidate impacted relation.
- *Req13 refines Req11 (Case 2 sub d)*: Modifying the predicates of Req11 may have an impact on the predicates of Req13 since there must be implication relation between refined and refining predicates of R_1 & R_2. Or the refines relation may not be valid anymore. Req13 is the candidate impacted requirement and *refines* is the candidate impacted relation.

The candidate directly impacted elements for *change 3* are Req13 and *requires* & *refines* relations. When we analyze the modification of Req11, we determine that Req13 is impacted and we should add the feature of sms messages sending to Req13. Refines and requires relations which we identified as candidate impacted relations are still valid for *change 3*. The new Req13 is the following: *The system shall allow lecturers to send e-mail and sms messages to students in the same group.*

The second step is to determine indirectly impacted elements by *change 3*. Since the only impacted requirement is Req13, we analyze the impacted elements for Req13:

- *Req13 requires Req12 (Case 3 sub c)*: Since the required requirement is not impacted by deleting the requiring requirement, *Req12* is not impacted. *requires* relation is the candidate impacted relation.

In our approach, we can eliminate Req10 as not impacted but this requirement should be checked in RequisitePro since we identify all requirements (Req10 and Req13) related to Req11.

6 Related Work

Several authors address change impact analysis in the context of requirements modeling. In [27], a metamodel and an environment based on requirements are described. The tool supports graphical requirements models and automatic generation of Software Requirements Specifications (SRS). Their tool supports checking constraint violations for requirements models. However, they do not give any formal definition for their requirements relations and they do not support change impact analysis upon requirements and their relations.

Some authors [9] [23] use UML profiling mechanism for goal-oriented requirements engineering approach. Heaven et al. [9] introduce a profile that allows

ISBN 978-82-14-04396-9

the KAOS model [26] to be represented in UML. They also provide an integration of requirements models with lower level design models in UML. Supakkul et al. [23] use UML profiling mechanism to provide an integrated modeling language for functional and non-functional requirements that are mostly specified by using different notations. None of these study the formalization of relations and change impact analysis for requirements.

Ramesh et al. [20] propose models for requirements traceability. Models include basic entities like *Stakeholder*, *Object* and *Source*. Relations between different software artifacts and requirements are captured instead of relations between requirements.

In [21], an approach is proposed to define operational semantics for traceability in UML, which capture more precisely the intended meaning of various types of traceability. They claim that it will enable richer tool support for managing and monitoring traceability by making use of consistency checking technology. They define the semantic property of a traceability relationship with a triplet (event, condition, actions). Although they do not focus on a specific domain, their results are valid for change impact analysis on requirements models. Walderhaug et al. [29] propose a generic solution for traceability that offers a set of services that is meant to cover both specification and appliance of traceability. Their solution is specified as a trace metamodel with guidelines and templates. Van Gorph et al. [25] illustrate the need for developer tolerance of inconsistencies. This motivates the use of fine-grained consistency constraints and a detailed traceability metamodel. They are interested in managing inconsistencies between different model artifacts. However, they do not provide any techniques to determine the impacts within a model. Albinet et al. [1] explain how to define requirements according to a proposed requirements classification and they present tracing mechanisms based on the SysML UML 2.0 profile. They describe their methodology in order to take into consideration the expression of requirements, and their traceability along the software life-cycle.

Maletec et al. [17] describe an XML based approach to support the evolution of traceability links between models. They use a traceability graph to detect the dependency between model elements. However, they do not discuss change impact analysis. Luqi [16] uses graphs and sets to represent changes. Ajila [2] explicitly defines elements and relations between elements to be traced with intra-level and inter-level dependencies. Impact analysis based on transitive closures of call graphs is discussed in Law [13]. We have the transitive closure for requires, refines and contains relations between requirements. Lindvall et al. [15] show tracing across phases again with intra-level and inter-level dependencies. They also discuss an impact analysis based on traceability data of an object-oriented system. However, they do not support their analysis with formalism.

Change impact analysis for software architectures has been studied by Zhao et al. [28]. They use a formal architectural description language to specify and graphs to represent the architectures. They restrict their analysis to the architectural level and not for analysis level.

ISBN 978-82-14-04396-9

7 Conclusion

In this paper, we proposed a change impact analysis technique based on formalization of requirements relations within the context of Model Driven Engineering. We gave a requirements metamodel with well defined types of requirements relations. This metamodel represents the common concepts extracted from some prevalent requirements engineering approaches. The requirements relations in the metamodel were used to trace related requirements for change impact analysis. Using the formalization of these relations allowed us providing proofs of more precise rules for change impact analysis.

We applied our approach in a case study based on a requirements specification for course management system. We were able to determine candidate impacted requirements and relations with a better preciseness. Since we applied the approach to a limited number of requirements, the results may not be very convincing. However, applying it to a number of requirements like 300 requirements will make the benefit of our approach more explicit. On the other hand, we are aiming at a tool that provides semi-automatic support for change impact analysis based on the presented rules. Such a tool may use a Prolog engine to notify the requirements engineer about candidate and actual impacted elements by using these rules.

Determining the impact of requirements changes on inferred relations in [7] with tool support is another future work in evolution dimension. For the evolution of requirements, we also want to analyze the impact of requirements changes in architectural and detailed design. We need trace models in order to link requirement models to design models. These trace models will enable us to determine possible impacts of requirements changes in design models.

References

1. Albinet, A., Boulanger, J.L., Dubois, H., Peraldi-Frati, M.A., Sorel, Y., Van, Q.D.: Model-Based Methodology for Requirements Traceability in Embedded Systems. ECMDA TW 2007 Proceedings, Haifa, 2007.
2. Ajila, S.: Software maintenance: An approach to impact analysis of object change. Software - Practice and Experience, 25 (10), 1155-1181, 1995
3. Bohner, S.A.: Software Change Impacts – An Evolving Perspective. Proceedings of the International Conference on Software Maintenance (ICSM'02).
4. Cleland-Huang, J. et al.: Event-Based Traceability for Managing Evolutionary Change. IEEE Transactions on Software Engineering. Vol. 29, issue 9, 2003.
5. Dean, M., Schreiber, G., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D., Patel-Schneider, P., Stein, L. A.: OWL Web Ontology Language Reference W3C Recommendation. (2004)
6. Gennari, J., Musen, A., Fergerson, R.W., Grosso, W.E., Crubezy, M., Eriksson, H., Noy, N.F., Tu, S.W.: The Evolution of Protégé: An Environment for Knowledge-Based Systems Development. International Journal of Human-Computer Studies, Volume 58, Issue 1, pp.89-123, January 2003
7. Goknil, A., Kurtev, I., van den Berg, K.: A Metamodeling Approach for Reasoning about Requirements. ECMDA-FA'08, LNCS, vol. 5095, pp. 311-326, 2008.

ISBN 978-82-14-04396-9

8. Guide to Software Engineering Body of Knowledge. IEEE Computer Society. http://www.swebok.org/ (last visit 06.02.2008)
9. Heaven, W., Finkelstein, A.: UML Profile to Support Requirements Engineering with KAOS. IEE Proceedings – Software, Vol. 151, No. 1, February 2004
10. Horrocks, I., Patel-Schneider, P., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: Semantic Web Rule Language – Combining OWL and RuleML. W3C, May, 2004
11. Jess, the Rule Engine for the Java Platform. http://herzberg.ca.sandia.gov/
12. Koch, N., Kraus, A.: Towards a Common Metamodel for the Development of Web Applications. ICWE 2003, LNCS, vol. 2722, pp. 497--506, Springer, Heidelberg (2003)
13. Law, J., Rothermel, G.: Whole program path-based dynamic impact analysis. International Conference on Software Engineering (ICSE`03), 2003
14. Lopez, O., Laguna, M.A., Garcia, F.J.: Metamodeling for Requirements Reuse. Anais do WER02 - Workshop em Engenharia de Requisitos, Valencia, Spain (2002)
15. Lindvall, M., Sandahl, K.: Traceability aspects of impact analysis in object-oriented systems. Software Maintenance: Research and Practice, 10, 37-57, 1998.
16. Luqi: A Graph Model for Software Evolution. IEEE Transactions on Software Engineering, 18 (8), 917-927.
17. Maletec, J.I., Collard, M.L., Simoes, B.: An XML based Approach to Support the Evolution of Model-to-Model Traceability Links. Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering, 2005.
18. Meyer, J.J.C., Wieringa, R., Dignum, F.: The Role of Deontic Logic in the Specification of Information Systems. Logics for Databases and Information Systems, pp.71-115 (1998)
19. OMG: SysML Specification. OMG ptc/06-05-04, http://www.sysml.org/specs.htm
20. Ramesh, B., Jarke, M.: Toward Reference Models for Requirements Traceability. IEEE Transactions on Software Engineering, Vol. 27, No. 1 (2007)
21. Reshef, N., Paige, R. et al.: Operational Semantics for Traceability. ECMDA TW 2005 Proceedings, Nuremberg, November 8th, 2005.
22. Sommerville, I.: Software Engineering. Addison-Wesley, 7th Edition, 2004.
23. Supakkul, S., Chung, L.: A UML Profile for Goal-Oriented and Use Case-Driven Representation of NFRs and FRs. SERA'05 (2005)
24. SPARQL Query Language for RDF. W3C, January 2008. http://www.w3.org/TR/rdf-sparql-query/
25. van Gorph, P., Altheide, F., Janssens, D.: Traceability and Fine-Grained Constraints in Interactive Inconsistency Management. ECMDA TW 2006 Proceedings, Bilbao, 2006.
26. van Lamswerdee, A.: Goal-Oriented Requirements Engineering: A Roundtrip from Research to Practice. Invited Minitutorial, Proceedings RE'01 - 5th International Symposium Requirements Engineering, pp. 249-263, Toronto (2001)
27. Vicente-Chicote, C., Moros, B., Toval, A.: REMM-Studio: an Integrated Model-Driven Environment for Requirements Specification, Validation and Formatting. In Journal of Object Technology, Special Issue TOOLS Europe 2007, Vol. 6, No. 9, pp. 437-454 (2007)
28. Zhao, J., Yang, H., Xiang, L., Xu, B.: Change impact analysis to support architectural evolution. Journal of Software Maintenance and Evolution: Research and Practice, 14, 317-333, 2002
29. Walderhaug, S., Johansen, U., Stav, E.: Towards a Generic Solution for Traceability in MDD. ECMDA TW 2006 Proceedings, Bilbao, 2006.
30. Warmer, J., Kleppe, A.: Object Constraint Language: The Getting Your Models Ready for MDA. Addison-Wesley, 2003.

ISBN 978-82-14-04396-9

Appendix: Requirements for Course Management System

Req1. The system shall allow end-users to provide profile and context information for registration.
Req2. The system shall provide functionality to search for other people registered in the system.
Req3. The system shall provide functionality to allow end-users to log in the system with their password.
Req4. The system shall support three types of end-users (administrator, lecturer and student).
Req5. The system shall allow lecturers to set an alert on an event.
Req6. The system shall maintain a list of events the students can be notified about.
Req7. The system shall notify the students about the occurrence of an event as soon as the event occurs.
Req8. The system shall actively monitor all events.
Req9. The system shall notify students about the events of the lectures they are enrolled for.
Req10. The system shall allow students to enroll for lecturers.
Req11. The system shall allow lecturers to send e-mail to students enrolled for the lecture given by that lecturer.
Req12. The system shall allow assigning students to teams for each lecture.
Req13. The system shall allow lecturers to send e-mail to students in the same group.
Req14. The system shall allow lecturers to modify the content of the lectures.
Req15. The system shall give different access rights to different types of end-users.
Req16. The system shall support two types of end-users (lecturer and student) and it shall provide functionality to allow end-users to log in the system with their password.

ISBN 978-82-14-04396-9

ISBN 978-82-14-04396-9

Traceability for Model Driven, Software Product Line Engineering

Nicolas Anquetil[1], Birgit Grammel[2], Ismênia Galvão[3], Joost Noppen[1,3], Safoora Shakil Khan[4], Hugo Arboleda[1], Awais Rashid[4], Alessandro Garcia[4]

[1] Ecole des Mines de Nantes, France
{nicolas.anquetil,johannes.noppen,hugo.arboleda}@emn.fr
[2] SAP Research CEC Dresden
birgit.grammel@sap.com
[3] University of Twente
i.galvao@ewi.utwente.nl
[4] Lancaster University, UK
{shakilkh,garciaa,marash}@comp.lancs.ac.uk

Abstract. Traceability is an important challenge for software organizations. This is true for traditional software development and even more so in new approaches that introduce more variety of artefacts such as Model Driven development or Software Product Lines. In this paper we look at some aspect of the interaction of Traceability, Model Driven development and Software Product Line.

Keywords: Product line, traceability, object-oriented, aspect-oriented.

1 Introduction

Traceability of artefacts elicits the means of understanding the complexity of logical relations and dependencies existing among artefacts that are generated during the software development lifecycle. Numerous kinds of artefacts are generated at the individual development stages, ranging from requirement artefacts to design elements down to source code fragments. With the inception of model-driven software development the scope of artefacts has been diversified by introducing models concerning, business processes, system requirements, architecture, design, tests, etc.

Since software development is ever facing the challenge to minimise development costs, advancing fields of Software Product Line (SPL) engineering and generative programming have been fostered. This in turn raises the need for more intricate traceability solutions, which in addition to classical end-to-end traceability, have to support for the traceability of variabilities and commonalities in the SPL. One of the main objectives of the European project AMPLE[1] is to bind the variation points in

1 http://ample.holos.pt/

ISBN 978-82-14-04396-9

various development stages and dimensions into a coherent variability framework across the SPL engineering life cycle thus providing forward and backward traceability of variations and their impact.

In this paper we present various perspectives of the AMPLE project on traceability for Model Driven, SPL engineering. The remainder of this paper is organized as follows. Section 2, introduces basic concepts of SPL engineering and contextualize traceability in it. Section 3, proposes a categorization of traceability links for SPL. Section 4, discusses how to deal with uncertainty and tracing the rational of decisions during the SPL development process. Section 5, looks at fine grained traceability links when mixing Model Driven development and SPL. Finally, Section 6 presents our conclusions and future work.

2 Software Product Line

The software industry is in crisis. It is unable to produce software at the pace required by the market: Projects are delayed, they fail to meet quality requirements, their budget is exceeded, expected functionalities are not delivered. SPL comes as an answer to this situation. It promises to deliver software faster, with higher quality and at a lower cost [1]. In this section we will introduce the basic concepts of SPL and how SPL and traceability interact.

2.1 Basic Concepts

The key to SPL promises (faster, better, cheaper) is to target, not a single system, but a family of similar systems all tailored to fit the wishes of a particular market from a constrained set of possible requirements. SPL is about producing software for a well defined market, from a base software architecture, with a predefined set of options, called variation points. To achieve higher quality more rapidly, it is based on reuse: of the software architecture and of the software components that may be plugged into it. Although the initial architecture and software components may be costly to develop, successive applications inside the family are cheaper and cheaper as they reuse most of what was build for the previous applications [1]. The SPL paradigm uses two processes and two main focuses (see Figure 1): (i) *Problem Space* focuses on defining what problem the family of applications, or an individual application in the family, will address; (ii) *Solution Space* focuses on producing the software components to solve that problem; (iii) *Domain Engineering* is responsible for establishing the software family platform, first by identifying typical requirements of the problem space and where they will be authorized to vary, second by developing the software architecture that address these requirements and the software components that fit into the architecture, and; (iv) *Application Engineering* is responsible for deriving individual applications from the requirements of a customer (inside the set of authorized variations) and composing this application from the family architecture and the available components.

Although the two were conceived separately, Model Driven Development is a

ISBN 978-82-14-04396-9

natural candidate to fit in the general framework of SPL: One may develop a meta-model that can be transformed in different applications according to the wishes of the customer [2,3]. The general solution is described in Figure 1. First, in domain engineering, one defines a meta-model of the problem (top left), specifying the concepts that may be used in the creation of a solution --an application--, and a feature model (model of all the features available). Second, still in domain engineering, but in the modelling of the solution (top right), one defines transformation rules to generate code from the application model (when it will be available). Obviously the future application model must conform to the meta-model of the product line. In addition, manually written software components (source code) can also be created that are intended to be combined with the automatically generated code later. Third, in application engineering (bottom left), one defines a model of a particular application (conforming to the meta-model defined in domain engineering). One also selects the features that should be implemented in this particular application. Finally, in the solution space (bottom right), the application is automatically derived from the model by applying the transformation rules.

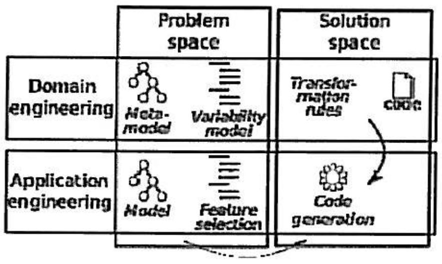


Fig. 1. How MDD fits in the two processes and two spaces of SPL.

2.2 Traceability for SPL

Traceability is recognized by all to be highly important for SPL engineering. On top of traditional concerns for traceability, SPL has to deal with variability and with two development processes. Variability is the description of all possible variation points in the family products, and all the variants, the available options for each variation point. Traceability appears as a key asset to manage this complexity. Variability is seen as the one fundamental aspect of SPL, and specific to it, that needs to be traced.

The difficulties linked to traceability in SPL are [4]: (i) there is a large number and heterogeneity of documents, even more than in traditional software development; (ii) one needs to have a basic understanding of the variability consequences during the different development phases; (iii) one needs to establish relationships between product members (of the family) and the product line architecture, or relationships between the product members themselves; and (iv) there is still poor general support for managing requirements and handling complex relations.

We could not find much tool support, neither available for industrial use nor in form of research prototypes in academia. Traceability means to link several artefacts

ISBN 978-82-14-04396-9

at different levels and the rationale of this link. One has to link documents, stakeholders and the rationale behind the links. Since software development and more specifically SPL development is a complex task one has to trace many objects of various kinds with different structures. In [5] a general presentation of the traceability needs and the integration in a SPL are proposed. The traceability requirements are: (i) it should be based on the semantics of models used in the SPL infrastructure; (ii) it should be customized to capture relevant trace types; (iii) it should be capable to handle variability; (iv) a small set of traces is better; and, (v) it should be automated when possible.

Berg *et al.* [6] view software engineering for single (traditional) systems in two dimensions, one for the development process and the other for levels of abstraction. All development artefacts can be placed somewhere in these dimensions. Variability adds a third dimension that explicitly captures variability information between product line members. This approach establishes a conceptual variability model which provides the appropriate mapping between all variation points in the two dimensional space (development process and levels of abstraction).

Ajila and Ali Kaba [7] use traceability to manage the SPL evolution. They identify three sources of changes in product line: (i) changes in an individual product; (ii) changes in the entire product line; and (iii) repositioning of an architectural component from individual product to the product line. The authors also analyse more precisely the reasons and the nature of changes in SPL development. The dimensions of analysis can be: motivations that led to the change (external or internal) and changes in the management process.

In [8] Moon and Chae propose a meta-modelling approach to trace variability between requirements and architecture in SPL. They define two meta-models for requirements and architecture integrating variability. These matrices contain information computed from the software structure and the variability points. Three kinds of relationships are provided: (i) between artefact constituents and trace matrix; (ii) between artefact constituents and models, and; (iii) between artefact constituents and specifications.

We will now present several propositions to treat traceability in Model-Driven SPL. Not all of them are specific to SPL, but all of them will be applied to this context in the AMPLE project.

3 Categorization of Traceability

As pointed out by the Center of Excellence for Traceability [9], the precise semantic of traceability links is poorly understood, and there is possibly a wide range of semantics. This is aggravated by the fact that it may not be desirable (or even possible) to create a closed set of semantic kinds of links. If one casts into stone the kinds of semantic links, then one loses flexibility for user-defined links that might be necessary to meet different project or company needs. But not predefining the link semantics, would greatly complicate automatic and elaborate treatment. We chose to develop a two layered solution with a high level abstract categorization, that we hope is general enough to fit all purposes; and a lower level, more detailed categorization,

ISBN 978-82-14-04396-9

that may be too specific in some specific situations. This is still a research issue and we have no definitive answer yet.

3.1 Dimension of Traceability

Traditional (non SPL) software engineering (e.g. [10, p.59], [11, p.526], [12]) defines two forms of traceability: vertical and horizontal. Unfortunately, different authors swap the respective definitions of vertical and horizontal! In this paper, we will call them intra and inter traceability. *Inter* traceability refers to relationships between different levels of abstraction: from requirements to models to implementation. *Intra* traceability refers to relationships between artefacts at the same level of abstraction: between related requirements, between models, between software components, etc. To this initial framework, SPL engineering introduces a third dimension, orthogonal to the two other ones to deal with *variability* and its implications (See also [6]). Traceability links are required to relate variation points (options) to their variants (choices), variants between themselves (when one choice constrains another one), variation points between themselves, low level artefacts to variation points or variants, and finally, choices made at the application engineering level to options offered at the domain engineering level. Finally, since dealing with configuration management is also a goal of the AMPLE project, we include a fourth dimension, *evolution*, for relationship between the various versions and revisions of a given artefact.
Note that there may be interactions between the different dimensions. For example, intra and inter traceability links may evolve between two versions of the SPL. This indicates that *intra* and *inter* traceability links may themselves be related by *evolution* traceability links. Variability traceability links are also subject to evolution over time. Finally, intra and inter traceability links may also be subject to variability traceability. For example, if two artefacts have an intra or inter traceability link in the domain model, and if both appear in the corresponding application model, then they should exhibit the same intra or inter traceability link in the corresponding application model. In summary, we may propose a hierarchy of dimensions: *Evolution* traceability may also apply to *intra*, *inter* or *variability* traceability relationships (and not only on artefacts). *Variability* traceability may also apply to *intra* or *inter* traceability relationships. All other interactions between two dimensions are considered meaningless.

3.2 Taxonomy of Traceability

There are quite a few approaches for inter and intra tracing intraditional systems [22],[23]. But these approaches do not fulfil the needs of SPL due to dependencies existing: (i) from core assets (domain engineering) to products (application engineering); (ii) between commonality and variability at different abstraction level; (iii) for core assets used by multiple products in a family of products. During the development of a SPL, numerous entities, artefacts, and models are created during both domain engineering and application engineering [16][17]. This makes it complex

ISBN 978-82-14-04396-9

to maintain and evolve the large number of intricate trace dependencies.

To facilitate trace maintenance and evolution in SPL, we propose to move away from simple associative trace links to links that capture the semantics of the relationship between the traced artefacts. We define a semantics-based dependency taxonomy wherein the dependency information: captures intricate information about the traces; promotes better understanding of the trace relationships; justifies the rationale for existence of a particular trace link, and; determines the significance of a trace link and help determine its consequence or impact on tracing information during SPL evolution. The taxonomy describing various facets of a dependency is influenced from conventional requirements engineering approaches, SPL concepts, and work on dependencies by [21]. We also investigated two case studies: HealthWatcher [18][19] and MobileMedia [20]. From these studies we structured the dependency links around two characteristics: *nature* and *granularity* .The *nature* of a dependency describes the fundamental categorization of the trace formed and helps define the significance of the dependency (which may vary from domain to domain) holding at the same level of abstraction (intra), higher to lower abstraction (inter), or between core assets and product(s). The nature of dependency can be categorized as: *Goal*, *Conditional*, *Service*, *Task*, *Temporal*, and *Infrastructure*. A more detailed discussion on *nature* of dependency taxonomy is presented in [19]. The *granularity* of a dependency elaborates on the trace by providing a better insight into the fundamental categorization of the trace (nature of dependency) formed at the same level of abstraction (intra), higher to lower abstraction (inter), or from core assets to products. The *granularity* of dependency helps identify the number of entities impacted directly or/and indirectly when a requirement, design, or implementation is evolved. The granularity of dependency can be categorized as: *Refinement*, *Composition*, *Constraint*, *Multiplicity*, *Behavioural*, and *Structural*.

We now discuss a brief trace scenario from the SmartHome industrial case study to showcase the dependency taxonomy. The SmartHome application bridges different technologies in a house like central heating, security system, household appliances through mobile phones and/or personal computers to retrieve the status, set or modify the control/setting of the devices. Our example scenario describes traces amongst the artefacts in application domain. The climate control system for managing the central heating ensures that the temperature a user (owner) has specified for the house is maintained. The desired temperature is maintained by automatically turning the central heating on/off when the specified temperature is reached. The nature of dependency for the requirement forms a (service, conditional) dependency with the HeatingComponents and Thermometer components at architectural level. Service and conditional dependency is formed as the Thermometer component gets the temperature of the house and the HeatingComponent turns the central heating on/off if the temperature is above or below the specified temperature range. The granularity of dependency is (behavioral) as the HeatingComponent reacts to the data output from the Thermometer component. The example shows the dependency model help extract end-to-end trace information between the loosely and/or tightly coupled requirements and architecture providing the system analyst an understanding of how requirements are being realized at architecture level.

ISBN 978-82-14-04396-9

4 Traceability in the Presence of Uncertainty

Independent of the categories of traceability and their nature and granularity, we propose to attach additional information to traceability links: The rational for its creation and the confidence we have in this rationale.

During software development, a large number of design decisions must be resolved. Typically, for each design issue several candidate solutions are considered. The rationale behind these design decisions is frequently based on assumptions made about diverse relevant criteria related to these candidates, calculating the alternatives' overall quality, and choosing the most appropriate solution. Ideally, the information used for taking such decisions would be of perfect quality, i.e. clear and accurate. However, in practice it is very difficult to attain accurate information at the moment it is required. As important decisions are taken in early phases of development, software architects will only have a partial and abstract view of the final, complete system. As a result, the design activities generally are performed with assumptions on relevant system characteristics that only partially provide the information with the desired quality. The rationale for design decisions is naturally subject to uncertainty.

Uncertainty plays a role in any system that needs to evolve continuously to meet the specified or implicit goals of the real world [13]. But while SPL engineering is based on the principles of reuse and variability management, the development of SPLs can suffer from uncertain information. As product line architectures are used over a prolonged period of time, they become subject to unforeseen evolution and maintenance. Moreover, the requirements definition and architectural design phases typically will be prone to uncertain inputs, as the product line is intended to support a versatile product family in volatile markets with changing demands. As a result of the variety of product families, the complexity of product line architectures and the longevity over which these must be maintained and evolved, it can be argued that the impact of uncertainty on product line development can be even more sever than traditional software systems.

As seen in section 2.1, the evolution of SPL artefacts in the problem space and the solution space, both in domain engineering and application engineering levels, can profit from model-driven techniques. The flexibility of model transformations offers ample means to address evolution of product lines. For example, model-driven approaches can automate the generation of trace links between source and target artefacts involved in a transformation [14]. Nonetheless, the application of MDE approaches does not resolve all problems caused by evolution and uncertainty in SPL development. MDE artefacts are subject to evolution. Change requests may cause the evolution of metamodels, models and model transformations. Moreover, the definition and realization of a model-driven approach can suffer significantly when uncertainty in the available information is not recognized and addressed accordingly.

Under this perspective, traceability of design decisions in SPL development is an important and relevant issue, as these are key points where uncertainty influences the design process. For performing traceability in the presence of uncertainty, the focus of handling uncertain information in particular should be on the rationale used to resolve design decisions. By identifying the uncertainty that exists in design decision rationale and modelling it accordingly in the decision process, its negative influence can be minimized. Further, tracing information on design decisions facilitates the

ISBN 978-82-14-04396-9

understanding of the impact of the uncertainty on the development of the SPL. Tracing the rationale of decisions improves the understanding of the important contextual factors that impact the quality of the SPL and variability management.

To this end, we have defined a meta-model that conceptualizes the kinds of design decision rationale in which we are interested, such as problem, alternatives, quality attributes, context and arguments. This meta-model comprises elements from argument-based rationale methods, problem-solving approaches and quality evaluation methods. Moreover, the meta-model accommodates the representation of uncertainty in the assumptions made by the developers while taking design decisions. Uncertainty is represented by utilizing techniques from fuzzy set theory.

The rationale behind each relevant design decision can be a model instantiated from the design decision rationale meta-model. Such models are themselves also considered as traceable artefacts. Therefore, the traces related to or from design decision rationale instances are stored along with inter or intra traceability relationships. For example, the design decision rationale can be traced to other decisions, or from and to other artefacts, such as requirements and architectural models. In this way, we are able to analyse the influences of uncertainty in the design rationale, while performing traceability for the sake of, for example, change impact analysis and root-cause analysis.

5 Traceability and Fine Grained Variability

We saw in Section 2.1 how, in the Model Driven, SPL approaches [2,3] a particular application is defined as a model conforming to the meta-model of the product line. The application must also choose available features from a feature model. By default, this approach does not allow fine grained selection of features, an application either has or not a feature. We call *large variation* a characteristic that affects the whole application [15]. For instance, properties such as localization (English or German), or, in a Smart Home system, a large variation could express that the house can have automatic lights (this would imply that *all* the lights in the houseare managed automatically). In contrast to this, we also define the concept of *fine variation* [15]. A fine variation is a characteristic that may be applied to specific elements of the application model. For example, in a Smart Home system, a fine grained variation could express that specific rooms of the house have the feature automatic light, but not all of them. Note that, large variations can be treated as special case of fine variation where all the elements of the model individually have the feature of interest.

The gain in flexibility of fine variation comes at the cost of more complex models and meta-model, with many new artefacts, model-to-model transformations, etc. Maintaining and evolving all the relations between individual elements and their features would require detailed management of traceability at a fine grained level. To manage this additional complexity, we defined a constraint models as part of the problem space modelling, during domain engineering [15]. This constraint model expresses what features may be linked (with fine variation) to what element of the meta-model. The constraint model also restrict the possible bindings by bounding possible cardinality and specifying properties that the element should have when

ISBN 978-82-14-04396-9

bound to the feature (for example, room that have automatic light requires some sensor). The actual binding of an element of the application to a feature occurs at the application engineering level, during the modelling of the problem. This is the moment where possible bindings described in the constraint model are created (or not) between actual elements of the application (concept in the model of the application) and the features offered in the feature model. Each binding defined in the application model is automatically checked against the restrictions expressed in the constraint model. The transformation of the application model to implementing code is realized by transformation rules.

Fine grained variability works in three dimensions of traceability: The specification of binding constraints between meta-concepts and features is an intra traceability, as both are at the same level of abstraction (during domain engineering, as part of the problem space modelling); binding of a concept to a feature is a variability traceability as the concepts appear during application engineering whereas the features are specified at the domain engineering level. Finally, the implementation of a given binding between a concept and a feature is an inter traceability.

6 Conclusion and Future Work

There is no doubt that traceability is a fundamental discipline of modern software development. As new development approaches emerge, such as Software Product Lines (SPL) engineering, the challenges of traceability, still not complete tackled, are increased. For example, SPL engineering increases the range of artefacts (variability model, variation points, variants). Model Driven Development (MDD) is another approach that also introduces new artefacts (meta-models, transformation rules).

In this paper, we looked at the AMPLE project, which is interested in the interaction of MDD and SPL with respect to traceability. We proposed a categorization mechanism for traceability links that offers to level of semantic: at the higher level we have four general traceability dimensions; at the lower level we propose finer grained semantic categories that may be specific to Model Driven, SPL engineering. We also discuss the problem of tracing development decision in the presence of uncertainty. Finally, we proposed a fine grained traceability mechanism between a domain meta-model and a product line variability model.

AMPLE is a project in progress and we started to implement these ideas in a traceability framework (described in another paper presented at this workshop). Other actions include creating industry case study to test our tools.

Acknowledgements. This work is supported by the European Commission grant IST-33710 - Aspect-Oriented, Model-Driven Product Line Engineering (AMPLE),

References

1. K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer Verlag, Heidelberg, 2005.

ISBN 978-82-14-04396-9

2. K. Czarnecki, M. Antkiewicz. "Mapping Features to Models: A Template Approach Based on Superimposed Variants". *GPCE'05.*, Lecture Notes in Computer Science, Vol. 3676, pages 422-437. Springer Verlag. 2005.
3. M. Voelter and I. Groher. "Product Line Implementation using Aspect-Oriented and Model-Driven Software Development". *SPLC'07*, pages 233-242
4. W. Jirapanthong and A. Zisman. "Supporting Product Line Development through Traceability". *APSEC'05*, pages 506--514. .
5. J. Bayer and T. Widen. "Introducing traceability to product lines", *Fourth International Workshop on Product Family Engineering (PFE-4)*, pages 399-406, 2001.
6. K. Berg, J. Bishop, and D. Muthig. "Tracing software product line variability - from problem to solution space", *SAICSIT '05*pages 182-191
7. S. Ajila and B. Ali Kaba. "Using Traceability Mechanisms to Support Software Product Line Evolution". *IRI'04*, pages 157-162.
8. M. Moon and H. S. Chae, "A Metamodel Approach to Architecture Variability in a Product Line", *, 9th International Conference on Software Reuse*, Lecture Notes in Computer Science vol. 4039, pages 115-126, Springer Verlag, 2006.
9. G. Antoniol et. al.: "Grand Challenges in Traceability". *Technical Report COET-GCT-06-01-0.9*, Center of Excellence for Traceability, September 2006.
10. J. O. Grady. *System Requirements Analysis. Academic.* ISBN: 978-0120885145, Academic Press, Inc., Orlando, FL, USA, 2006.
11. S. L. Pfleeger. *Software Engineering: Theory and Practice.* ISBN: 0131469134 Prentice-Hall, Inc., 3rd edition, 2005.
12. S. Ambler, "What's up with agile requirements traceability?", Dr.Dobb's portal, http://www.ddj.com/architect/184415807, Oct. 18, 2005 (consulted Apr. 18, 2008).
13. M. M. Lehman and J. F. Ramil. "Software Uncertainty". *Soft-Ware 2002: Computing in an Imperfect World*, Lecture Notes in Computer Science vol. 2311, pages 477–514. Springer Verlag, 2002.
14. I. Galvão and A. Goknil, "Survey of Traceability Approaches in Model-Driven Engineering". *EDOC'07*, pages 313-326,.
15. H. Arboleda, R. Casallas, and J.-C. Royer. "Dealing with Constraints during a Feature Configuration Process in a Model-Driven Software Product Line". *DSM'07*, pages 178–183.
16. S. D. Kim, S. H. Chang and H. J. La. "Traceability Map: Foundations to Automate for Product Line Engineering". SERA 2005, pages 340-347
17. W. Jirapanthong and A. Zisman. *XTraQue: traceability for product line systems.* Journal: Software and Systems Modeling. Springer Berlin/Heidelberg, 2007
18. P. Greenwood et. al. "On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study". ECOOP'07, pages 176-200
19. S. S. Khan, et. al.: "On the Interplay of Requirements Dependencies and Architecture". CAiSE'08
20. E. Figueiredo, et. al.: "Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability", ICSE'08
21. B. Ramesh and M. Jarke. "Towards Reference Models for Requirements Traceability". IEEE Transactions on Software Engineering. 27(1), Jan 2001.
22. S. Ajila. *Software Maintenance: An Approach to Impact Analysis of Object Change*, Software Practice and Experience, Vol. 25, No., pp. 1155-1181, 1995.
23. S. Ibrahim et al.: *A Requirements Traceability to Support Change Impact Analysis.* Asian Journal of Information Tech. 2005, Vol. 4, No. 4, pages 345-355

ISBN 978-82-14-04396-9

Supporting the Evolution of Software Product Lines

Ralf Mitschke and Michael Eichberg

Software Technology Group
Department of Computer Science
Technische Universität Darmstadt, Germany
{mitschke,eichberg}@st.informatik.tu-darmstadt.de

Abstract. Traceability enables the maintenance and evolution of software product lines as a whole as well as of specific products.
In general, traceability during software product development can be achieved by storing all product related artifacts in a configuration management system. But, the available systems are designed for single software products and are not able to support engineering of multiple products based on product lines. Current systems do not offer traceability that enables comprehensive reasoning about the evolution of the relationships between artifacts, associated features and the derived products. To enable traceability, we propose to use feature models as the primary means to organize and structure the artifacts of software product lines (SPLs). In our approach, each version of an artifact is associated with a specific version of a feature and feature dependencies are explicitly managed. By making a product line's feature model and product configurations first class entities of a configuration management system comprehensive traceability in SPLs is enabled.

1 Introduction

Software product line (SPL) engineering facilitates production of families of applications [1]. In SPLs features that are common to multiple products are identified and managed. The dependencies between the features are also captured and describe constraints on the combinations of features. The features are then used in the implementation of different concrete products, where each product represents a valid combination of features. The goal of SPLs is to enable a controlled reuse and evolution of features common to multiple products.

An open challenge when developing SPLs is to enable controlled evolution of product lines [2]. Changing requirements will require adaptation of a product line's features and, hence, affect deployed products as well as products still under development. In general, successful evolution of SPLs requires comprehension of the impact of changes. For example, if a product has a (security) bug, it is necessary to identify all products (deployed and under development) that are affected. Also, if a feature is added or removed it is necessary to assess the impact on all products, e.g., to determine if an already deployed product can be maintained

ISBN 978-82-14-04396-9

using the current product line or if an older version of the product line needs to be used. Hence, comprehensive traceability and support for reasoning about changes in SPLs is required to determine how changed requirements affect existing products. Existing approaches only support the management of artifacts and configurations for single applications. These systems do not support reasoning about changes based on the evolution of features, their interdependencies and the dependencies to products.

To enable comprehensive reasoning about the relations between features, products and a product line as a whole, we propose to use feature models at the core of configuration management systems for SPLs; i.e., to use feature models to structure a SPL's artifacts. Feature models [3] are an established notation to specify a product line's commonalities and variabilities, but were not yet used as part of configuration management systems.

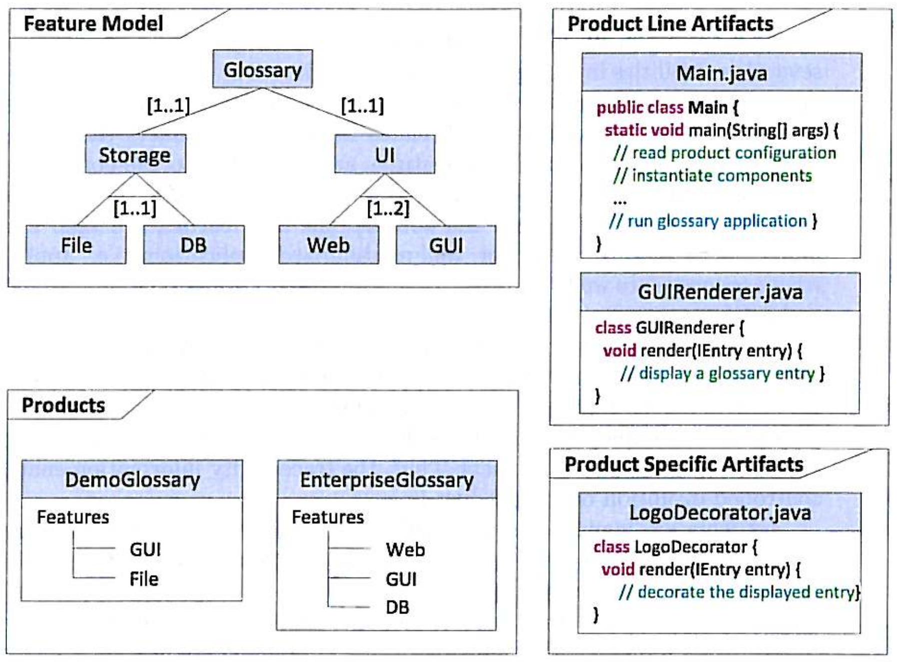
The remainder of this paper is organized as follows. In the next section, we discuss a small product line used to exemplify the following discussions. In Section 3, we discuss versioning of SPLs using feature models. In Section 4, related work is presented. Section 5 summarizes this paper.

2 Product Line Evolution

To illustrate problems that arise during software product line evolution, we present a small product line and discuss its evolution. The product line is used to build *glossary* products, i.e., applications for managing lists of terms and their definitions. The glossaries vary w.r.t. data storage techniques and the method of user interaction. A glossary is either stored in a database or using a flat file. User interaction with the glossary is possible using a desktop GUI or a web-based interface. Based on the product line, several products can be built, e.g., a `DemoGlossary` and an `EnterpriseGlossary`. Figure 1 depicts the feature model of the product line, the features of the products, and an excerpt of the development artifacts that realize the product line.

The feature model notation we use is cardinality based [4], e.g., the `UI` feature in Figure 1 has a feature group that allows the selection of one or two of its child features. The product line is implemented using a component based approach, in which each feature is implemented by a component that can be deployed individually. In general, different techniques can be used to implement variabilities and bind specific implementations in the products [5, 6]. In our case, the variabilities are bound in the `Main.java` class during application startup. The corresponding product's configuration file is read and the components that implement the required features are instantiated. The same technique is used to implement product specific extension. For example, the `EnterpriseGlossary` provides a mechanism for branding the glossary with a company's logo, which is then shown next to the title of each entry. This is achieved by instantiating a `LogoDecorator` component that uses adaption mechanism provided by the underlying framework to add the logo.

ISBN 978-82-14-04396-9



```
public class Main {
  static void main(String[] args) {
    // read product configuration
    // instantiate components
    ...
    // run glossary application }
}
```

```
class GUIRenderer {
  void render(IEntry entry) {
    // display a glossary entry }
}
```

```
class LogoDecorator {
  void render(IEntry entry) {
    // decorate the displayed entry}
}
```

Fig. 1. Overview of the glossary product line

Now, we consider a change scenario, in which the specification of the glossary product line is extended to include user authentication. This change is reflected in the feature model through an `Authentication` feature, that is added as an optional sub-feature of the `UI` feature. This change impacts: (1) the implementations of the `GUI` and the `Web` features. These features must offer respective authentication screens that ask for user credentials; (2) product specific implementations, for all products that incorporate the additional feature. Assuming, for example, that the `EnterpriseGlossary` incorporates the `Authentication` feature. As a result, the product's logo decoration mechanism must be validated against the new `GUI` and `Web` implementations.

This example demonstrates that changes of the feature model can directly influence the implementation of features as well as products. Traceability is required to ensure that the product line as a whole remains consistent in the presence of such changes.

3 Feature-Driven Versioning

To support software product line evolution and maintenance, we propose to use feature models [3, 7] to organize the artifacts of a product line w.r.t. the features they realize. In general, features do not always correspond to artifacts in a one-

ISBN 978-82-14-04396-9

to-one mapping [6, 8]. For example, conditional compilation approaches encode several variabilities in a single source-code artifact. Our feature-driven versioning approach is based on the assumption, that a mapping between features and artifacts depending on multiple features is possible. Artifacts that implement functionality related to multiple features are associated to the common parent of these features. In the glossary product line, for example, the artifacts that encode functionality for both the `Web` and the `GUI` feature, are then managed under the `UI` feature. Moreover, well modularized applications, i.e., applications where concerns are well separated, naturally support a mapping between features and artifacts.

To enable feature-driven versioning, we have developed a configuration management system with a version model, that enables versioning of features, products and artifacts. Furthermore, the model relates these versions to enable traceability. For example, changes in the feature model are assessed regarding their impact on products and artifacts. Thus, the traceability information enables the controlled evolution of the product line.

Artifacts are managed by associating each feature/product with an artifact repository. Inside each repository a versioned container, denoted as *feature/product container* manages all artifacts that belong to a specific feature/product version. Inside such a container artifacts are developed incrementally using the check-in/check-out facilities provided by traditional configuration management systems. For example, the `Main.java` file in Figure 1, is checked out, a bugfix is implemented, and then checked in again, all inside a single version of a feature container. A container's version is incremented whenever breaking changes are made in the specification of the underlying feature/product. For example, the addition of an `Authentication` feature, as discussed in the previous section, increments the container versions for the `Web` and `GUI` features, since they must include respective user authentication screens.

Figure 2 depicts the version information of the glossary product line in it's initial state as presented in Figure 1. A feature's version information is composed of the *logical version* and the *container version*. The logical version denotes the number of evolutionary steps that affected a feature. The container version denotes the version of a feature's container. The resulting versioning model incorporates the following version information:

A **Feature logical version**: The generation of a feature w.r.t. affecting feature model changes.
B **Feature container version**: The overall state of the functionality provided by a feature.
C **Artifact version**: State of an artifact associated to the respective feature container.

In the following, we discuss our versioning model in detail w.r.t. all involved entities, i.e., features, artifacts, and products. In addition to features and products our approach also supports versioning for feature groups and product groups. Feature groups form the connection between parent and child features

ISBN 978-82-14-04396-9

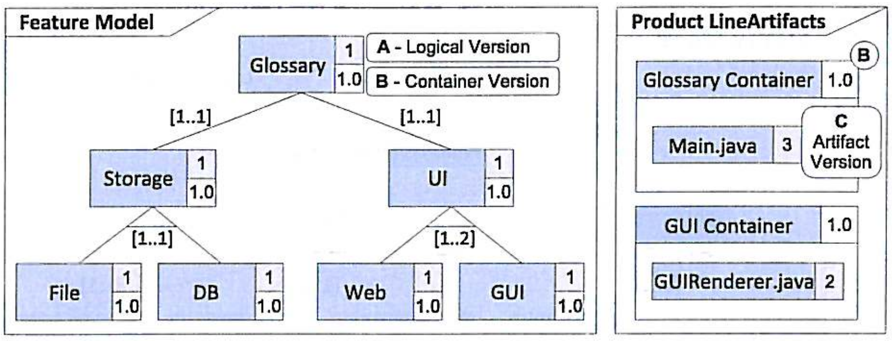


Fig. 2. Initial versioning of the glossary product line

in feature modeling. A product group denotes a set of products that share specific features. For example, the set of products that share the DB feature denotes all products, which specifically include the DB feature and leave the rest of the variability open. This approach facilitates a multi-step configuration process for products, where each step becomes more explicit concerning the included features [7]. In feature-driven versioning, we allow artifacts to be associated with features, feature groups, products and product groups. However, in the following discussion, we only discuss the association between artifacts and features, since the other allowed associations are treated in the same manner.

3.1 Feature Logical Versioning

In our approach, we version each feature w.r.t. the feature model. This means that a feature's version changes logically, if we add, remove, or change child features. A child feature is connected to the parent feature through a reference in a feature group. Therefore, we have defined a versioning mechanism, that versions feature groups, and propagates version increments from a feature group to the parent feature. Each feature contains one or more feature groups, and the logical version of a feature is incremented if a feature group is added, removed, or if the logical version of a contained feature group is incremented. Accordingly, a feature group's version is incremented, when changes occur in a referenced feature or if the cardinality constraint for valid feature selections changes.

In addition to changes resulting from child features, the logical version is incremented, if the specification of a feature changes. This also results in a new feature container version, which is discussed in detail in the next section.

The version model always propagates increments of logical versions up to the root feature of the feature model. Figure 3 depicts the increments in logical versions for adding an `Authentication` feature as described in the introductory change scenario. Note, that the logical versions of features (groups) are incremented automatically.

ISBN 978-82-14-04396-9

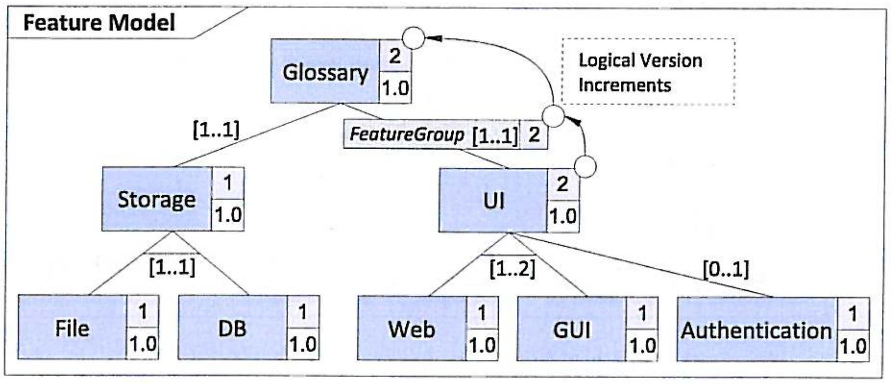


Fig. 3. Incrementation of feature's logical versions

The proposed versioning model enables reasoning about a feature's change in isolation, i.e., it is possible to identify how often a feature has changed and when a feature has changed without requiring an understanding of the evolution of the product line as a whole. This provides local reasoning, in the sense that, for example, the changes in the `UI` feature increment the `Glossary` feature, while the subtree of the `Storage` feature remains unaffected. Especially, if different stakeholders are responsible for the implementation of a set of features, they can see which features have changed and whether their subtree was affected. This information is immediately recognizable, regardless of the size of a subtree, since the version of the topmost feature in the subtree remains the same.

3.2 Feature Container Versioning

The feature container version denotes the version of a feature w.r.t. the specification of its functionality. Whenever the specification is changed, the implementation provided for a feature must be adapted. Hence, the respective feature container version must be incremented. Changes in a feature's specification may be directly related to changes in the feature model, but can also occur independently. For example, the specification of the `GUI` feature can change and prescribe the use of a new widget library. Each increment of a feature container version also increments the logical version of a feature, which propagates to the logical version of the root feature, thus denoting that the SPL as a whole has changed.

When a container version is incremented, the contained artifacts are virtually copied to the new version of the container. A trace link is kept to provide traceability of the artifact versions across different feature container versions.

Figure 4 depicts a version increment of the `GUI` feature's container. The changes to contained artifacts are illustrated exemplarily, by copying the artifact `GUIRendered.java` to the new container version and resetting the artifacts version information to the initial value of `1`. In the feature model, the new

ISBN 978-82-14-04396-9

container version results in incremented logical versions for the `GUI`, `UI` and `Glossary` features.

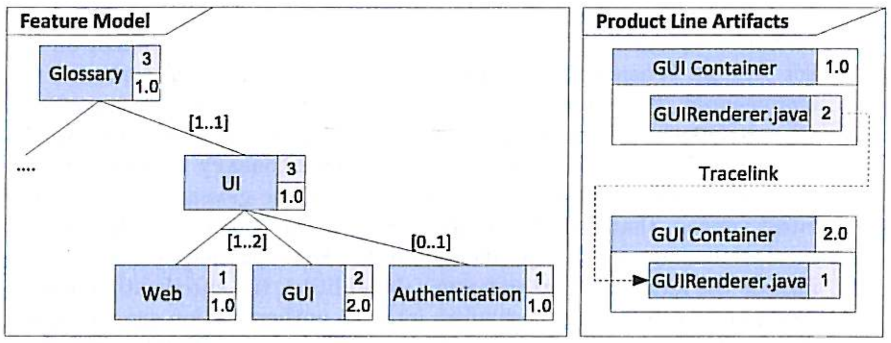


Fig. 4. Incrementation of feature's container versions

The container approach to versioning a feature's artifacts provides the basis for SPL maintenance. Let's assume that an error must be corrected in the initial implementation of the `GUI` feature. The traceability information can be used to identify all products in all versions that include the initial version or successor versions of the `GUI` feature. Thus, all relevant products can be identified, fixed and then redeployed.

3.3 Product Versioning

Support for product versioning is necessary to enable comprehensive traceability between product versions and the features' versions used in the implementation of a product.

A products version reflects the specification of a product. The specification determines the included features and their versions, as well as the product specific functionality. Hence, if during the development of a product the set of selected features changes or if a feature's logical version changes the product's version is updated. If these changes are not relevant for a product, it's version is not affected. Additionally, a product's version is updated when the specification of the product changes; i.e., if its functional or non-functional requirements change.

To identify the versions of selected features, a product is associated with the logical version of the feature model's root feature. Since version increments in the feature model always propagate up to this root feature, the version of the root feature identifies all other feature versions. Thus, no recordkeeping for every version of every feature selected in the product is required. Associating a product with the root feature's version then expresses, that the product is valid for the constraints in the respective feature model version.

ISBN 978-82-14-04396-9

To trace which product versions were deployed and delivered to customers each product is associated with a deployment version. The deployment version identifies which product version was delivered and when it was built.

To store a product's specific artifacts a product is also associated with a versioned container. The container's version is directly dependent on the product version. Whenever a new product version is created its container version is incremented. The containers are linked to determine which version of an artifact was used as the foundation for the development of a later product version.

Figure 5 depicts the changes of the `DemoGlossary` and the `EnterpriseGlossary` products from the initial version of the glossary product line to the latest version, that includes the `Authentication` feature and the related implementation changes. The specification of the `EnterpriseGlossary` is changed to include the `Authentication` feature. In addition, the specification is changed to incorporate a branding mechanism into the authentication screens, thus resulting in a total of two increments of the product version. Furthermore, the initial `EnterpriseGlossary` version was deployed as version `1.0`, while the changed product was deployed as version `2.0`.

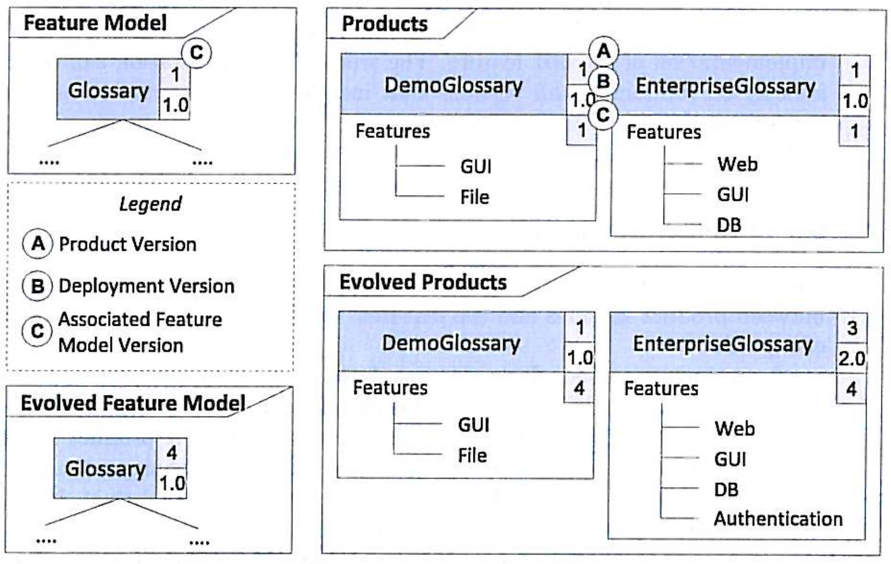


Fig. 5. Incrementation of product versions

3.4 Realization

Our proposed feature-driven versioning approach is realized based on the Subversion[9] configuration management system. All artifacts related to one fea-

ISBN 978-82-14-04396-9

ture/product are managed by one Subversion repository. Each version of a feature/product container is mapped to one Subversion branch. On top of Subversion we have implemented the necessary logic to relate feature and product versions, to manage their dependencies, and to relate them to artifact repositories. This enables traceability between features, products and artifacts.

4 Related Work

The evolution of software product lines is already discussed in [10–13]. However, none of them focus on feature models in particular nor on combining versioning models with traceability between features, products and artifacts.

In [10] the authors propose to utilize traceability information for supporting software product line evolution. The authors argue about the necessity for traceability relationships in software product lines in general and propose to use impact analysis to determine a generalized change set. This change set is to be monitored by interested stakeholders to assess the risk of changes for the software product line.

Using configuration management systems to provide adequate support for software product lines has been the topic of [11–13]. In [11] the author presents the specific requirements of configuration management for product lines. In the works of [12] and [13] it is proposed to utilize configuration management systems as the basis for an automated product derivation process. Their approaches envision systems, that construct products from common artifacts and allow to change these artifacts in the implementation of products. These changes can be later reincorporated to the common artifacts in the product line on an automated basis.

Advanced approaches in software configuration management systems allow feature-based configuration building [14], i.e., selection of artifacts and configurations based on system features such as supported operating system. These approaches do not account for the larger context of software product lines, where changes also occur to the features and their relationships.

5 Conclusion

In this paper, we have presented a versioning model that enables traceability between features, products and artifacts in the context of software product line evolution. Our feature-driven versioning approach can be used as a basis for managing evolving software product lines. The provided traceability information ensures consistency and maintainability of software product lines.

Acknowledgments

This work is supported by European Commission grant IST-33710: Aspect-Oriented, and Model-Driven Product Line Engineering (AMPLE), 2006 - 2009.

References

1. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)
2. Svahnberg, M., Bosch, J.: Evolution in software product lines: Two cases. Journal of Software Maintenance **11**(6) (November 1999) 391–422
3. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented Domain Analysis FODA Feasibility Study. Technical report (1990)
4. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice **10**(1) (2005) 7–29
5. Van Gurp, J., Bosch, J., Svahnberg, M.: On the notion of variability in software product lines. In: WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01), Washington, DC, USA, IEEE Computer Society (2001)
6. Gacek, C., Anastasopoules, M.: Implementing product line variabilities. SIGSOFT Softw. Eng. Notes **26**(3) (May 2001) 109–117
7. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multilevel configuration of feature models. Software Process: Improvement and Practice **10**(2) (2005) 143–169
8. Muthig, D., Patzke, T.: Generic Implementation of Product Line Components. In: NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World, London, UK, Springer-Verlag (2003) 313–329
9. CollabNet, I.: Subversion (2007) http://subversion.tigris.org/.
10. Ajila, S.A., Kaba, A.B.: Using traceability mechanisms to support software product line evolution. In: Information Reuse and Integration, IRI 2004, IEEE (November 2004) 157–162
11. Krueger, C.W.: Variation Management for Software Production Lines. In: SPLC 2: Proceedings of the Second International Conference on Software Product Lines, London, UK, Springer-Verlag (2002) 37–48
12. Laqua, R., Knauber, P.: Configuration Management for Software Product Lines. In: 1. Deutscher Software-Produktlinien Workshop, Fraunhofer IESE (2000)
13. van Gurp, J., Prehofer, C.: Version management tools as a basis for integrating Product Derivation and Software Product Families. In: 10th Software Product Line Conference. (2006)
14. Zeller, A., Snelting, G.: Unified versioning through feature logic. ACM Trans. Softw. Eng. Methodol. **6**(4) (October 1997) 398–441

ISBN 978-82-14-04396-9

A Model-Driven Traceability Framework to Software Product Line Development

André Sousa[1], Uirá Kulesza[1], Andreas Rummler[2], Nicolas Anquetil[3], Ralf Mitschke[4], Ana Moreira[1], Vasco Amaral[1], João Araújo[1]

[1] CITI/DI/FCT, Universidade Nova de Lisboa, Portugal,
als12171@fct.unl.pt, {uira,amm,ja,vasco.amaral}@di.fct.unl.pt
[2] SAP Research, Dresden, Germany,
andreas.rummler@sap.com
[3] Ecole des Mines de Nantes, INRIA, France,
nicolas.anquetil@emn.fr
[4] TU Darmstadt, Germany,
mitschke@st.informatik.tu-darmstadt.de
WWW: http://www.ample-project.net

Abstract. In this paper, we present a model-driven traceability framework to software product line (SPL) development. Model-driven techniques are adopted with the aim to support the flexible specification of trace links between different kinds of SPL artefacts. A traceability metamodel is defined to support the flexible creation and storage of the SPL trace links. The framework is organized as a set of extensible plug-ins that can be instantiated to create customized trace queries and views. It is implemented based on the Eclipse platform and EMF technology. We illustrate a concrete instantiation of the framework to support the tracing between feature and use cases models.

Key words: Product lines, multi-agent systems, object-orientation, aspect-orientation, traceability.

1 Introduction

Software product lines (SPLs) methods and techniques [1–4] aim at producing software system families with high levels of quality and productivity. A system family [5] is a set of programs that shares common functionalities and maintain specific functionalities that vary according to specific systems being considered. A software product line (SPL) [1] can be seen as a system family that addresses a specific market segment. In order to improve the productivity and quality of SPL development, the proposed methods and techniques motivate the specification, modeling and implementation of a system family in terms of its common and variable features. A feature [2] is a system property or functionality that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in SPLs.

ISBN 978-82-14-04396-9

The development of SPLs [6] is typically organized in terms of two main processes: domain and application engineering. The domain engineering focuses on: (i) the scoping, specification and modeling the common and variable features of a SPL; (ii) the definition of a flexible architecture that comprises the SPL common and variable features; and (iii) the production of a set of core assets (frameworks, components, libraries, aspects) that addresses the implementation of the SPL architecture. In application engineering, a feature model configuration is used to compose and integrate the core assets produced during the domain engineering stage in order to generate an instance (product) of the SPL architecture.

Despite the advantages and benefits of current SPL methods and techniques, most of them do not provide automatic mechanisms or tools to address the traceability between the produced artefacts in both domain and application engineering processes. This is fundamental to guarantee and validate the quality of SPLs development and to allow a better management of SPL variabilities. On the other hand, current traceability tools do not provide support to address the new artefacts (variability models) or processes (domain and application engineering, product management) of SPL development. As a result, they do not allow the explicit management of SPL common and variable features along all the development process.

In this context, this paper proposes a model-driven traceability framework for SPL development. Our framework aims to support forward and backward tracing of SPL artefacts using model-driven engineering techniques. It proposes to support the automatic management and maintenance of trace links between SPL artefacts of domain and application engineering. It is implemented as a flexible framework in order to allow its customization to different SPL traceability scenarios.

The remainder of this paper is organized as follows. Section 2 gives an overview of a survey of existing traceability tools developed in the context of a joint project. Section 3 details our model-driven traceability framework to software product line development by presenting the traceability metamodel adopted, the framework implementation architecture and the instantiation of the framework to support the tracing between features and use cases. Section 4 discusses implementation issues and further steps of the framework development. Section 5 presents related work. Finally, Section 6 concludes the paper.

2 Analysis of Existing Traceability Tools

A survey on existing traceability tools was conducted in the context of the AMPLE project [7]. The objectives of this survey were to investigate the current features provided by existing tools in order to assess their strengths and weaknesses and their suitability to address SPL development. The tools were evaluated in terms of the following criteria: (i) management of traceability links; (ii) traceability queries; (iii) traceability views; (iv) extensibility; and (v) support for Software Product Lines (SPL) and Model Driven Engineering (MDE). We

ISBN 978-82-14-04396-9

believe that these criteria are crucial for this kind of tools since they provide the basic support to satisfy traceability requirements (creation of trace information and querying it) and other important concerns regarding the evolution of these tools and SPL development. These are explained in detail as follows.

The management of traceability links criterion was adopted to analyze the capacity of each traceability tool to create and maintain trace links (manual or automatic) and which kind of trace information is generated. The traceability queries criterion analyses which searching mechanism to navigate over the artefacts and respective trace links is available from the tools, varying from simple queries to navigate over the related artefacts to more sophisticated queries that support coverage analysis and change impact analysis. The traceability view criterion characterizes the supported views (tables, matrix, reports, graphics) that each tool provides to present the traceability information between artefacts. The extensibility criterion evaluates if any tool offers a mechanism to extend the tools functionalities or to integrate with any other software development tools. Finally, the support for SPL and MDE criterion indicates if a tool adopts any mechanism to aid in the development of software using these new modern software engineering techniques.

The conclusions that were drawn from our survey were that none of the investigated tools had built-in support for SPL development, and a vast majority of them are closed, so they cannot be adapted to deal with the issues raised by SPL. The surveyed tools were also not developed for the Eclipse platform, and only a few had some sort of mechanism to support software development in that environment [8, 9]. TagSEA is the only tool that was developed as an Eclipse plug-in, allowing the developer to insert specific tags in source files providing some traceability support. However, TagSEA is still in an experimental state and does not cover traceability up to a degree that is desirable.

There is some recent progress in providing traceability support for product lines. Two of the leading tools in SPL development, *pure::variants* [10] and GEARS [11] have defined some extensions to allow integration with other commercial traceability tools. *Pure::variants* includes a synchronizer for CaliberRM and Telelogic DOORS that allows developers to integrate the functionalities provided by these requirements and traceability management tools with the variant management capabilities of *pure::variants*. Similarly, GEARS allows importing requirements from DOORS, UGS TeamCenter, and IBM/Rational RequisitePro.

However, these *external* tools (e.g. DOORS, RequisitePro) handle traceability for traditional systems. Apart from their individual weaknesses (see Table 1), they all lack the ability to deal explicitly with specificities of SPL development, for example, dealing with variability. They do not provide advanced and specific support to deal with change impact analysis or requirement/feature covering in the context of SPL development.

Table 1 summarizes some key aspects of the evaluation of the tools that may be integrated with *pure::variants* or GEARS. In terms of trace links management, the tools only allow defining them manually, but they offer the possibility to import them from other existing documents, such as, MS-Word, Excel, ASCII

ISBN 978-82-14-04396-9

and RTF files. CaliberRM and DOORS allow the creation of trace links between any kinds of artefacts. RequisitePro focuses only on the definition of trace links between requirements.

The RequisitePro provides functionalities to query and filtering on requirements. CaliberRM allows querying requirements and trace links. DOORS provides support to query any data on the artefacts and respective trace links. Regarding advanced query mechanisms, Caliber RM allows detecting some inconsistencies in the links or artefacts definition, and DOORS offers impact analysis report and detection of orphan code. The traceability tools offer different kinds of trace views, such as, traceability graphical tree and diagram, and traceability matrix. All of them also allow navigating over the trace links from one artefact to another.

In terms of extensibility, CaliberRM allows specifying new types of reports and DOORS allows creating new types of links and personalized views. The three tools also provide support to save and export trace links data to external database through ODBC. DOORS integrates with many other tools (design, analysis and configuration management tools). None of the investigated tools has explicit support to address SPL, MDD or AOSD technologies.

	RequisitePro	**CaliberRM**	**DOORS**
(i) Links Management	Manual	Manual	Manual + Import
	Between requirements	Complete life-cycle	Complete life-cycle
(ii) Queries	Query & filter on requirements attributes	Filter on requirements & links	Query & filter on any data (including links)
	–	Links incoherence	Impact analysis, orphaned code
(iii) Views	Traceability matrix, traceability tree	Traceability matrix, traceability diagram, reports	Traceability matrix, traceability tree
(iv) Extensible	–	–	Creation of new type of links
	Trace data saved w/ ODBC	Trace data saved w/ ODBC	Integrates w/ > 25 tools (design, text, CM, ...)
(v)SPL and MDE	Not Supported	Not Supported	Not Supported

Table 1. Summary of the comparison of three requirement traceability tools according to our evaluation criteria (see text for explanation)

3 A Model-Driven Traceability Framework to SPL

In this section, we present and discuss the main topics regarding our SPL traceability framework. We initially describe the traceability metamodel adopted by

ISBN 978-82-14-04396-9

our framework (Section 3.1). Next the framework's architecture is presented, as well as the class diagram for the main modules (Sections 3.2 and 3.3). Finally, an instantiation of the framework allowing the management of trace links between features and use cases is also shown (Section 3.4).

In this section, we present and discuss the main topics regarding our SPL traceability framework. The framework's architecture is presented, as well as the class diagram for the main modules. Finally, an instantiation of the framework allowing the management of trace links between features and use cases is also shown.

3.1 Traceability Metamodel

The traceability metamodel which is the basis of the framework discussed in this article is shown in Figure 1. It is centered on the assumption that all trace information can be represented by a directed graph.

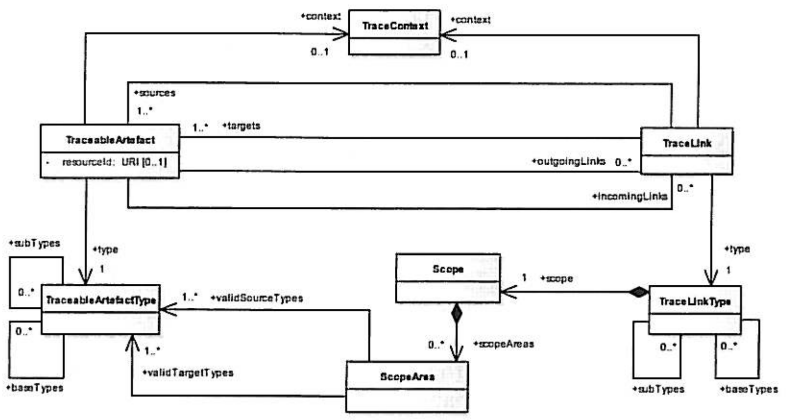


Fig. 1. Traceability Metamodel

The main elements of the metamodel are the following:

- A *TraceableArtefact* represents a (physical) artefact that plays a role in the development cycle. The granularity of such artefact is arbitrary, it may represent a requirement, an UML diagram, an element inside a diagram, a class or a method inside a class. An artefact is unambiguously identified by a locator (resourceId), which describes where this artefact is located (such as in a file or a directory) and how it may be accessed.

  TraceLink is the abstraction for the transition from one artefact to another. An instance corresponds to a hyperedge linking two artefacts in the trace graph. A transition is always directed; therefore a from-to-relation between artefacts is created by a trace link (between source and target artefacts).

ISBN 978-82-14-04396-9

- During the process of tracing information about the design of a software system, different artefacts of different types must be taken into account. For this reason each TraceableArtefact has an instance of *TraceableArtefactType* assigned. This type separates artefacts from each other. Artefact types may be grouped in a hierarchical manner, which mimics the concept of multiple inheritance, known from object orientation.
- Analogous to the type of an artifact, each link has a type because the relationship between two artefacts may differ. Examples for such types would be *contains*, *depends of* or *is generated from.* For this reason each instance of TraceLink is assigned to an instance of *TraceLinkType.*
- The existence of an artefact or the relationship from one artefact to another may be justified in some way. Not all artefacts and transitions would require such a justification, for example a "contain" transition is rather self explanatory. The attachment of additional information to artefacts and links can be modeled by attaching a *TraceContext* to relations and/or artefacts.
- Links of a certain type may only be valid between artefacts of a certain type. A link of type "contains" may be valid between a *Method* and a *Class*, but not between two *Architectural Models.* The narrowing of validity area of link types is modeled via the introduction of the elements *ScopeArea* and *Scope.*

3.2 Traceability Framework Overview

Our traceability framework aims to provide an open and flexible platform to implement trace links between different artefacts from SPL development. In order to address this aim, the framework is being designed and implemented based on the use of model-driven techniques. The traceability metamodel, described in Section 3.1, allows specifying different kinds of trace links between SPL artefacts. All the trace links stored must follow the guidelines established by this metamodel. The framework requires the definition of a variability model to allow the tracing of SPL common and variable features along the domain and application engineering stages. The variability model is used in our approach as the main reference to trace the SPL artefacts. However, the design of the framework is generic, so it may be applied outside SPL development.

The following main functionalities are provided by our framework to support the tracing of SPL artefacts: (i) creation and maintenance of trace links between a variability model and other existing artefacts (UML models, source code, etc); (ii) storage of trace links using a repository; (iii) searching of specific trace links between artefacts using pre-defined or customized trace queries. Trace queries can be executed over the trace links in order to select interesting traceability information to help the SPL development or evolution; and (iv) flexible visualization of the results of trace queries using different types of trace views, such as, tree views, graphs, tables, etc.

The architecture of the proposed traceability framework is being defined in terms of four main modules. Each of them is directly responsible to implement the framework main functionalities. Figure 2 shows the traceability framework architecture with its respective modules:

ISBN 978-82-14-04396-9

1. **Trace Register** - this module provides mechanisms to create and maintain (update, remove and search) trace links between a variability model and other artefacts;
2. **Trace Storage** - defines the storage mechanisms to persist the trace links between SPL artefacts;
3. **Trace Query** - this module allows to create and submit queries to search specific trace links previously stored; and
4. **Trace View** - it is used to specify visual representation of trace links between artefacts resulted from a submitted trace query.

Figure 2 also shows how the different framework modules are connected. The Trace Register and Query modules use the services provided by the Trace Storage module to store and search the trace links between SPL artefacts. The Trace Query module can invoke basic trace link queries methods provided by Trace Storage. Each trace query returns a set of trace links of interest between the artefacts under tracing. After the execution of a trace query, the Trace Query calls the Trace View module to allow the visualization and navigation over the resulted trace links and respective artefacts.

3.3 Traceability Framework Structure

The traceability framework is structured as an object-oriented framework that defines an infrastructure to provide basic services to search and store trace links and it also offers a set of extension points to create specific SPL traceability functionalities (trace queries and views). Figure 2 shows the general structure of the framework. The *TraceRegister*, *TraceQuery* and *TraceView* abstract classes represent the extension points of the framework main components depicted as UML packages. Each of them must be instantiated and customized to address specific traceability scenarios in SPL development. Next we give an overview of these framework classes.

The *TraceRegister* abstract class must be specialized to create specific ways to create and store trace links between artefacts. The `executeRegister()` abstract method is implemented with this purpose. The trace links are stored using the persistence mechanisms provided by the Trace Storage module. The framework does not specify the concrete ways that the trace links must be obtained. This functionality can be provided, for example, by specifying a strategy to automatically identify possible trace links between artefacts or by providing a graphical interface (e.g. check boxes) that enables the SPL developers to create explicitly the trace links. The *FeaturesExtractor* and *ArtefactsExtractor* abstract classes must also be specialized to provide specific ways of extracting features and other desired software artefacts. For instance, if the user wants to extract requirements from a certain requirements modeling tool, then an appropriate class inheriting from *ArtefactsExtractor* must be created and the corresponding `getSoftwareArtefacts()` abstract method must be implemented, which will be responsible for parsing the input file and extracting the desired elements.

ISBN 978-82-14-04396-9

The *TraceQuery* abstract class establishes the general structure to implement traceability queries. It uses the services from the Trace Storage module to search trace links of interest. After that, it delegates the resulted trace links from its query to an associated trace view class by calling the `showResults()` method. The trace views are implemented as subclasses of the *TraceView* class. This mechanism of submitting a query and presenting the results in the chosen view is implemented by the method `executeQuery()` in the *TraceQuery* abstract class. The current implementation of the Trace Storage module provides basic traceability services to retrieve and query basic trace links between specific artefacts. Our proposal in the framework is to create more advanced traceability queries (such as, requirements/feature coverage, change impact analysis, product variants tracing) from these basic ones.

3.4 Framework Instantiation: An Example

In this section, we present an instantiation of our framework that addresses the tracing between feature and use case models. Our aim is to illustrate how the framework can be used and extended to address concrete scenarios of traceability in SPL development. All the framework extension points, presented in Section 3.2, are illustrated in this instantiation.

The feature and use case models used to specify the commonalities/variabilities and requirements of a SPL were created using the following plug-ins: (i) the Feature Modeling Plug-in (FMP) [12] - that allows to create feature models and (ii) the MoPLine tool [13] - a model-driven tool being developed in the context of the AMPLE project to support the process of domain analysis of SPLs.

Figure 2 shows the classes codified during the process of instantiation of our traceability framework. Two concrete extractor classes (*FMPFeaturesExtractor*, *MoPLineUseCasesExtractor*) were defined to get the information about the use cases and features defined for a specific SPL. These extractors parse the model files produced by the FMP and MoPLine plug-ins and retrieve the list of features and use cases encountered there. Figure 2 also shows the *FeatureToUseCaseTraceRegister* class that represents an instance of the *TraceRegister* abstract class. It is used to create trace links between features and use cases.

Figure 3(a) shows the visual representation of the *FeatureToUseCaseTraceRegister*. As we can see, it presents the features and use case (and respective steps) that were collected from the FMP and MoPLine models. The *FeatureToUseCaseTraceRegister* allows defining specific trace links between the SPL feature and entire or partial steps of use cases. After the software engineer defines the trace links, this information is persisted using the services provided by the Trace Storage module.

In this framework instantiation scenario, we created different trace queries and associated views. Figure 2 shows two subclasses of *TraceQuery*: *TraceByFeatureQuery* and *TraceByProductVariantQuery*. The first one is used to trace the use cases that are connected to a specific feature. The other one is used to obtain the set of use cases related to a set of features that represents a product from a

ISBN 978-82-14-04396-9

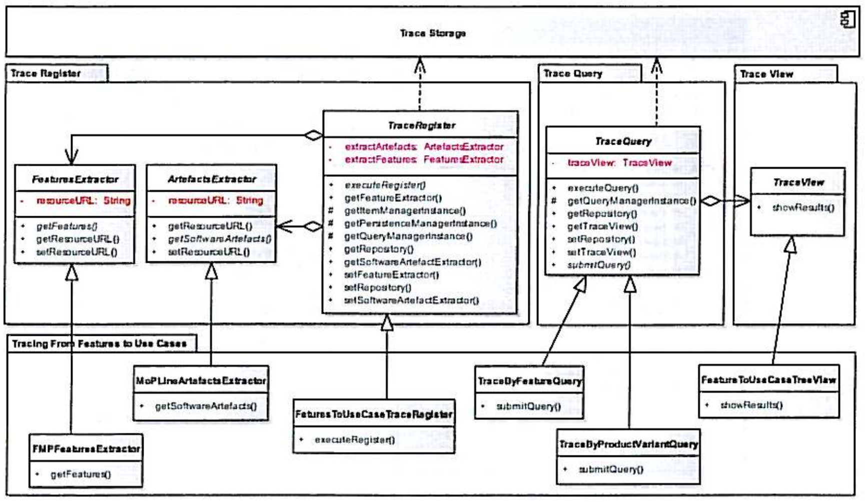


Fig. 2. Traceability Framework Architecture and Instantiation Scenario - Tracing from Feature to Use Case

SPL. Both these trace queries can be associated to the *FeatureToUseCaseTreeView* class that represents a specialization of the *TraceView* class.

Figure 3(b) shows the visualization of an instance of the *FeatureToUseCaseTreeView* that shows the results of tracing a set of features representing a product. The different use cases associated to those features is showed in the tree view graphical component. Although in this framework instantiation example, we have only illustrated the tracing from features to use cases, it is also possible to define trace queries and views to show the tracing from use cases (or other SPL development artefact or model) to features.

4 Implementation Issues and Further Steps

In this section, we present and discuss preliminary lessons learned from our development experience of the SPL traceability framework.

4.1 Framework Implementation on Top of the Eclipse Platform

The Eclipse has been chosen as the main platform to implement our traceability framework. The justification behind is easily explained: an established MDD-based technology already exists in this environment, called Eclipse Modeling Framework (EMF) [14]. The EMF provides a base platform for model-driven development. On top of this framework, several reusable components that are interesting in this context are also provided, such as, EMF Query, EMF Search and Teneo.

ISBN 978-82-14-04396-9

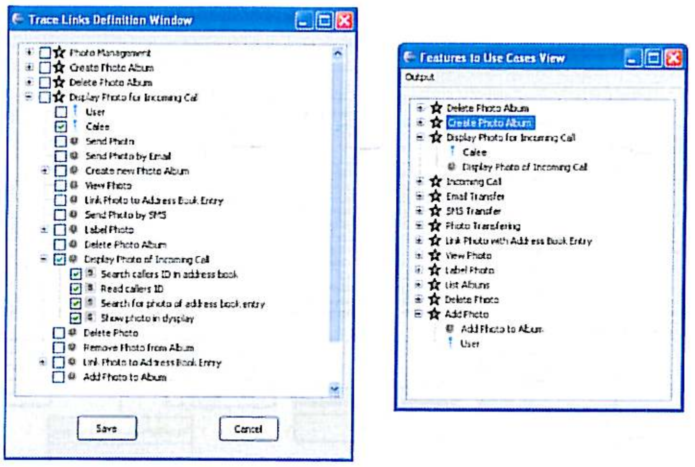


Fig. 3. (a) FeatureToUseCaseTraceRegister class and (b) FeatureToUseCaseTreeView

The traceability metamodel which is at the core of the framework has been implemented using EMF. It is a representative implementation of the OMGs EMOF standard. Other technologies used, obviously, have to be integrated with EMF.

A traceability framework has to be able to handle arbitrary sized sets of data - for this reason, it is fundamental to be supported by a database. Eclipse Teneo is used as the abstraction layer between the EMF and the actual database layer to persist EMF model instances. Teneo already supports storage, caching and retrieval of EMF object, although the layer is currently at version 0.8, which means it has not reached its final release state.

Queries on top of EMF models are formulated using EMF Query, which provides a Select-From-Where mechanism that allows basic statements to be executed quite easily. This provides the basis for more advanced queries explained in the next section.

On top of the implementation of the Trace Storage module, a collection of UI services can be provided for users and developers. In section 3.3, we have illustrated specific instances of the *TraceRegister*, *TraceQuery* and *TraceView* classes that were used to trace from features to use cases. We are currently organizing these specific subclasses to define a set of customizable classes that facilitates the process of development of new register, query and views. An example is the implementation of a tree view that allows tracing from features to any other SPL artefact. It is being defined based on our development experience on the tree view presented in Figure 3(a).

ISBN 978-82-14-04396-9

4.2 Advanced Traceability Queries

Basic traceability queries, as explained in section 3 are only a part of functionality required. Although basic traceability queries may be used to answer simple questions (i.e. which artefacts have been derived directly from a certain artefact), users are most likely not interested in such queries. Rather, more advanced questions derived from the concrete traceability real cases reside in the foreground. In this context, we are currently working in three kinds of advanced queries that are of particular interest of the AMPLE project:

- *Requirements/Feature Coverage*: query that some certain feature is really covered in the applied architecture and/or in the source code of the system. This could also be extended to test cases;
- *Change Impact Analysis*: discover possible side effects when changing a certain artefact. Such analysis is interesting in forward and backward direction: What would be the result of a change of some requirement on the actual architecture? What would be the impact on some feature of the substitution of a certain component by another one?
- *Product Variants Tracing*: perform a trace to all included artefacts in a certain product. This is closely related to Orphan Analysis, where artefacts should be discovered that are included in some product variant, which provide functionality that is not really needed in that variant.

We rely on the assumption, that such advanced queries may be composed entirely of a set of basic queries. While basic queries may be expressed in a dedicated query language, a frontend for the advanced query module may be a dedicated user interface guiding a user through the querying process, thus providing an easier access to the application.

5 Related Work

We have previously discussed the results presented in a traceability tools survey (Section 2). The conclusions drawn from this survey were that the existing traceability tools do not provide a sufficient support to address the traceability problem in SPL development. Even though some extensions have been developed, and integration with existing SPL tools, the functionalities provided by these tools are not sufficient for SPL development. They lack specific queries for product lines, which combine variability information with other dimensions of software development. For instance, there is no support for detection of feature interactions (a common problem in SPL) or tracing of product variants. The fact that the majority of these tools are closed for external development, and therefore cannot be adapted or extended, makes them incapable of completely satisfying the needs of software product lines.

We have presented a proposal for a traceability framework to address traceability in software product lines. Our approach is based on the work by Pohl et al. [4], which consists of establishing trace links between variability elements and

ISBN 978-82-14-04396-9

software artifacts. The major benefit of our framework is that all the software product line models (variability model, requirements model, etc) are variation points in our approach. The user can select which models to use during development, without being imposed with a specific metamodel. Our framework facilitates the ability to trace from features to any other artifact of software development (requirements, code, etc). This is achieved by using the traceability metamodel (Section 3.1), to store all the necessary trace information. This metamodel is generic enough to be applicable to a variety of scenarios, including SPL development. On top of the services provided by this repository we have defined a set of abstract classes and interfaces to allow the instantiation of our framework for different SPL scenarios.

6 Conclusions and Future Work

In this work, we presented a model-driven framework to SPL traceability. The main aim of our framework is to support forward and backward tracing of SPL artefacts using model-driven engineering techniques. We have been developing our framework in the context of the AMPLE project.

We presented our proposal for a traceability framework to address traceability in for product lines. Finally, we illustrated the current architecture and implementation of the framework, and provided an example of an instantiation that supports the definition of trace links between features and use cases. Implementation, open issues and further development steps were also discussed.

We are currently working in the evolution of the framework to address other different scenarios of traceability in SPL development, such as: (i) tracing from features to architectural, design and implementation models/artefacts; (ii) analysis of feature interactions; and (iii) feature covering and change impact analysis.

Acknowledgements: The authors are partially supported by EU Grant IST-33710: Aspect-Oriented, Model-Driven Product Line Engineering (AMPLE).

References

1. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston, MA, USA (2002)
2. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration using feature models. In Nord, R., ed.: Proceedings of the 3rd International Software Product Line Conference (SPLC 2004). Volume 3154., Springer, Boston, MA, USA (2004) 266–283
3. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley (2004)
4. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag, New York,USA (2005)
5. Parnas, D.: On the design and development of program families. In: IEEE Transactions on Software Engineering. Number 2. IEEE (1976) 1–9

ISBN 978-82-14-04396-9

6. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison Wesley Longman (2000)
7. AMPLE: Project AMPLE: Aspect-Oriented, Model-Driven Product Line Engineering. `http://ample.holos.pt`
8. Esterel Technologies: Scade suite. `http://www.esterel-technologies.com/products/scade-suite`
9. Telelogic: Telelogic doors. `http://www.telelogic.com/products/doors/index.cfm`
10. pure-systems GmbH: pure::variants. `http://www.pure-systems.com/Variant_Management.49.0.html`
11. BigLever Software Inc.: Software product lines - biglever software. `http://www.biglever.com`
12. Antkiewicz, M., Czarnecki, K.: FeaturePlugin: Feature Modeling Plug-In for Eclipse. In: Proceedings of the Workshop on Eclipse Technology eXchange (OOPSLA 2004), ACM, Vancouver, BC, Canada (2004) 67 – 72
13. Universidade Nova de Lisboa: Di-fct/unl: Mopline tool. `http://ample.di.fct.unl.pt`
14. Budinsky, F., Brodsky, S., Merks, E.: Eclipse Modeling Framework. Pearson Education (2003)

ISBN 978-82-14-04396-9

ISBN 978-82-14-04396-9

Application of Tracing Techniques in Model-Driven Performance Engineering

Mathias Fritzsche[1], Jendrik Johannes[2], Steffen Zschaler[2], Anatoly Zherebtsov[3], and Alexander Terekhov[3]

[1] SAP Research CEC Belfast & Queen's University Belfast
United Kingdom
mathias.fritzsche@sap.com & mfritzsche01@qub.cu.uk

[2] Technische Universität Dresden
D-01062, Dresden, Germany
jendrik.johannes@tu-dresden.de, steffen.zschaler@tu-dresden.de

[3] XJ Technologies Company
195220 St. Petersburg , Russian Federation
anatoly@xjtek.com, talex@xjtek.com

Abstract. In our previous work we proposed Model-Driven Performance Engineering (MDPE) as a methodology to integrate performance engineering into the model-driven engineering process. MDPE enables domain experts, who generally lack performance expertise, to profit from performance engineering by automating the performance analysis process using model transformations. A crucial part of this automated process is to give performance prediction feedback not based on internal models, but on models the domain experts understand. Hence, a mechanism is required to annotate analysis results back into the original models provided by the domain experts. This paper discusses various existing traceability methodologies and describes their application and extension for MDPE by taking its specific needs into account.

1 Introduction

Model-Driven Engineering (MDE) is a technique for dealing with the ever-increasing complexity of modern software systems. It places models of software—often expressed using domain-specific languages (DSLs)—at the heart of the development process. This enables developers to view and design a software system from a much higher level of abstraction than the code level, allowing them to cope with much higher levels of complexity. Additionally, the use of DSLs allows domain experts to be involved in the development of a software systems. This can increase the quality of software as the domain requirements can be taken into account more directly and accurately. For example, the authors of [1] describe how DSLs can be used to develop so called Composite Applications that access services provided by a SAP Business Suite system. This is supported by industrial tools, such as the Composition Environment (CE) [2] that applies MDE

ISBN 978-82-14-04396-9

for the development of Composite Applications, enabling experts in a domain to build new applications based on pre-provided modules.

However, a difficulty with MDE lies in supporting extra-functional requirements in the software system. As generic solutions that guarantee certain non-functional properties under any circumstance are typically difficult to provide, developers need expertise regarding specific non-functional properties and how to support them in application design. This expertise is typically lacking with domain experts. Additionally, the high-level of abstraction beneficial to the development of complex applications, can also make it difficult to provide reasonable estimates for non-functional properties of the resulting system.

Therefore, there is a need for a better support for non-functional properties within MDE. Performance is one such important property, which has been researched in the context of MDE [3,4] and is also the focus of this paper. We have previously proposed Model-Driven Performance Engineering (MDPE) [5], an extension of MDE that allows performance analysis models to be derived from development models at each level of abstraction. However, so far, the results of such an analysis still require performance engineering expertise to be interpreted. In particular, the performance engineer must understand the specific analysis or simulation technique used and be able to translate back the results from this analysis into properties of the original development models. In this paper, we investigate how this feedback of results can be automised in the context of MDPE, such that domain experts can benefit from analysis results without consulting performance engineers.

Trace information about all of the various transformations that together make up MDPE is the most important asset required for implementing result feedback. Therefore, in this paper, we will discuss various approaches to collect and maintain such trace information. We will then discuss which of these techniques is most appropriate in the context of MDPE and show how we have applied it to implement result feedback for MDPE. The contribution of this paper is, therefore, twofold: a) It presents a technique for feeding performance analysis results back into original development models, and b) to the tracing community it presents a case of application of tracing and a discussion of the benefits and drawbacks of a number of tracing techniques in a specific application context.

The remainder of the paper is structured as follows: We begin in Section 2 with a brief overview of MDPE including a description of where tracing information is required. Then, in Section 3 we describe the implementation of the feedback mechanism and also discuss which tracing technique is most suitable for this purpose. Section 4 describes related work and Section 5 concludes the paper.

2 Background

Performance engineering is used in software development to meet performance requirements in the design of a software system. Applying performance engineering is, however, costly since it requires performance experts, who understand the

ISBN 978-82-14-04396-9

formalisms that performance analysis and simulation tools use, to be consolidated. For this reason, it is often neglected or only done in the very beginning of system design. Consequently, the performance is only measured on the running system—which often leads to redesigns and reimplementations of (parts of) the system.

In Model-Driven Engineering (MDE), software is designed stepwise, by refining models until the concrete implementation is realized. Model-Driven Performance Engineering (MDPE) [5] proposes to do performance engineering at each of these refinement steps to discover design flaws early in the development process. Furthermore, it proposes to use model-driven techniques to automate performance engineering itself. To this end, we propose a semi-automatic generation of performance models based on development models (e.g. UML models) using model transformations. To have a sufficient cost-benefit this is also a stepwise process: basic analysis can be done automatically on each refinement level while more detailed analysis requires manual input to the generated performance models and therefore more performance expertise. Thus, MDPE takes two orthogonal dimensions of refinement into account: One dimension to refine the performance models, and another dimension to refine the development models in a traditional MDE process.

To define a process independent of development and performance analysis formalisms, we use a tool-independent performance metamodel. Development models from the MDE process are transformed into an instance of this metamodel: a Tool-Independent Performance Model (TIPM). Such a TIPM can be transformed into different performance analysis models called Tool-Specific Performance Models (TSPMs). These models are then employed by specific performance analysis tools using the same performance view-point on the system as common data base.

The TIPM offers a solution that is independent of any specific performance modelling concept, such as layered queuing models [6], stochastic petri nets [7], etc. Hence, an MDPE user is able to compare the capabilities of several performance modelling concepts without undergoing the error prone and time consuming task of defining the interfaces to the development modelling language in use [8]. Additionally, MDPE is independent of the performance analysis tool actually used, such as AnyLogic, etc., which simplifies the industrial application of MDPE. Finally, as a result of the TIPM we are able to support multiple kinds of development models, such as UML models, but also proprietary models used within SAP for the purpose of business behaviour modelling.

In [5] we presented a transformation from UML models to AnyLogic simulation models. This paper concentrates on the opposite direction: the tracing of results, collected by running the simulation models, to the UML models. To support this, the TIPM metamodel contains the concepts of monitors that can be filled with analysis results.

An excerpt from the metamodel is shown in Figure 1. The left side of the figure defines concepts that hold information about the structure of the studied system. These concepts—*Scenario*, *Step*, *PathConnection*, *Resource* and refine-

ISBN 978-82-14-04396-9

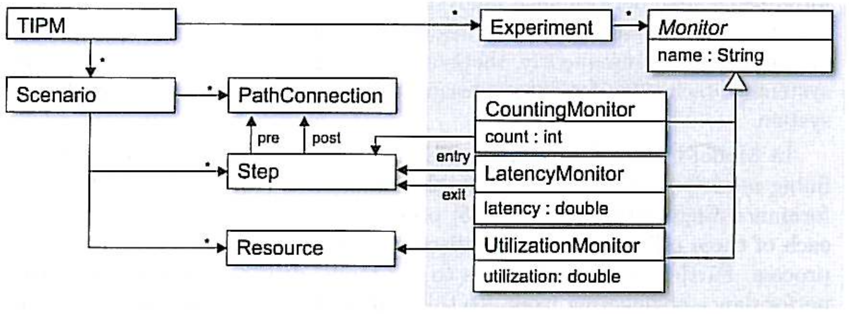


Fig. 1. The TIPM metamodel defines concepts to describe a system (left) and analysis results (right).

ments of those (not shown in the figure)—are based on the Core Scenario Model (CSM) introduced by Woodside et al. [9] and have basically the same semantics as defined there. The right side contains the concept of *Experiments* and *Monitors*, in addition to the concepts borrowed from the CSM. Those are used to indicate which kind of performance analysis should be performed and where. For the latter, different kinds of monitors can refer to different kinds of elements in a TIPM, which they observe. Their properties are only filled by the utilised analysis tool *after* an analysis has been performed.

In the figure, three different monitor types are defined: A *LatencyMonitor* holds information about the latency between two steps (*entry* and *exit*). A utilization measured for a resource can be placed into a *UtilizationMonitor*. A *CountingMonitor* observes how often a step is executed.

Like the transformation to a performance model, the tracing from a performance model is also a two-step process. In the first step, the simulation tool provides data to fill the monitors of a TIPM. Then this information can be used to update the development models from which that TIPM was generated. The following section discusses both steps in detail.

3 Extending MDPE with Traceability

Figure 2 provides an overview of our proposed architecture to extend MDPE with traceability. As shown in the figure, two steps, named as *Tool2TIPM Tracing* and *TIPM2DevelopmentModel Tracing*, are required in order to implement synchronization between performance analysis tools and development models. A description of both steps is provided in the following subsections.

3.1 Synchronization between Performance Analysis Tools and TIPMs

The tracing of simulation results back to a TIPM concentrates on filling the properties of monitor elements in the TIPM (cf. right side in Figure 1). These are

ISBN 978-82-14-04396-9

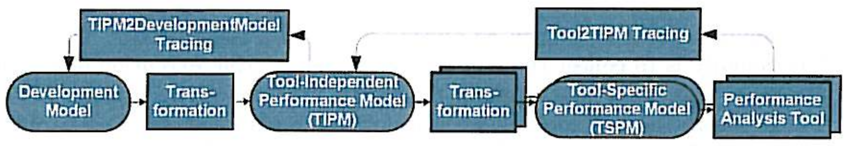

TIPM2DevelopmentModel Tracing
Tool2TIPM Tracing
Development Model
Trans-formation
Tool-Independent Performance Model (TIPM)
Trans-formation
Tool-Specific Performance Model (TSPM)
Performance Analysis Tool

Fig. 2. Tracing architecture for MDPE as Block Diagram [10]

initially not set, because they are explicitly provided as containers for feedback information. The performance analysis tool, which is responsible for providing the result data, has to know about the monitors and their properties. This has to be taken into account, when transforming a TIPM into a TSPM.

As an example of such performance analysis tool we used AnyLogic developed by XJ Technologies [11]. It is a multi-method simulation tool, which includes basic services that can be used to create simulation models using different methods—*discrete-event*, *system dynamics* or *agent-based modeling*—and allows combining these methods in one model. The object-oriented model design paradigm supported by AnyLogic provides modular and incremental construction of large models. The simulation engine is based on Java technology, which makes it possible to use functionality provided by the Java runtime library in simulation models.

AnyLogic supports developing custom object libraries that can include model objects developed with the tool itself together with Java objects and third-party libraries written in Java. An AnyLogic library can be attached to a model development environment and its objects can be used in other simulation models. In order to support simulations based on TIPMs, we developed a special AnyLogic library that includes objects which behave corresponding to concepts from the TIPM metamodel and collect data about the simulated model during its executions. To generate AnyLogic simulation models, two transformations were developed using the Atlas Transformation Language (ATL) [12]. The first one converts a TIPM into a structure of AnyLogic library objects as anticipated by AnyLogic. It generates all required objects together with additional objects required to connect everything into a working model. This structure includes all AnyLogic objects and connections between them that have to be present in the model. The second transformation applies XML formatting to make the structure readable by the AnyLogic tool, effectively leaving the MDE technology space.

To enable the actual feedback, we have implemented a small service to which a simulation tool like AnyLogic can send information. In this way, the simulation tool itself does not require any MDE specific knowledge. It is sufficient to send a message to a designated port containing the information which TIPM (identified through its filename) and which property in which monitor to fill (addressed through their unique names). When receiving such a message, the service updates the corresponding TIPM with the provided information.

ISBN 978-82-14-04396-9

Monitors are also defined as objects in our AnyLogic library. Their main functionality is to collect the required result data during execution of the simulation model. Type and scope of the information collecting is defined in the TIPM: Parameters of monitors are transformed to parameters of library objects together with information on how to connect to the service listening for result data. As mentioned, the AnyLogic simulation engine can use a wide variety of features provided by Java; This was used to realise the connection to the service. After a model's execution, AnyLogic connects to the specified port and provides result data. Assuming, for instance, that AnyLogic has measured a latency of *11.38 ms* for a certain sequence of steps, it can set the *latency* value of the corresponding *LatencyMonitor* to `11.38`.

3.2 Synchronization between TIPM and Development Models

For the tracing between development models and TIPMs a solution is required to trace between two modeling languages where one, defined by the TIPM metamodel, is known, but the other, used for defining the development models, may vary. Our current MDPE prototype, however, only supports UML models as development models. In the future we require support for other (domain-specific) languages, such as SAP proprietary languages for business process modeling as shown in [1], as well.

In the following, different options to implement tracing between development models (of arbitrary metamodels) and TIPMs are analysed and one is selected. Afterwards, we exemplify the actual feedback process on UML development models using the chosen traceability methodology.

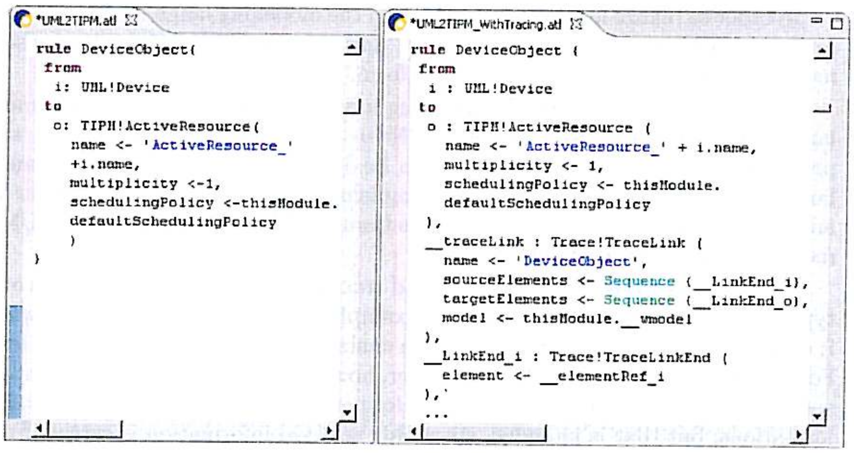
A straightforward option is the usage of bi-directional transformations, such as provided by the Query View Transformations (QVT) [13] relations language. Initial tool support has been published in [14]. We claim that this solution is tracing by design—in the sense that there is no need to care about tracing after the implementation of a transformation. However, it is more effort to develop bi-directional transformations than uni-directional ones as the transformation developer always has to keep both directions in mind. Thus, bi-directional are not an option for MDPE because we do not want to complicate the development of transformations between development models and TIPMs.

As another option, the transformation developer can provide a definition of how tracing information between development models and TIPMs are established after the transformation was performed. The Epsilon Comparison Language (ECL) [15] enables comparison of models of arbitrary metamodels. Hence, a transformation developer could use ECL to write a comparison specification using ECL that identifies correspondences between a development model and a TIPM. The disadvantage of this approach is that it currently requires manually writing comparison rules for each single transformation or, in other words, for each type of development model that should be supported. An approach supporting the definition of ECL rules in parallel with defining the transformations, could significantly reduce reduce the indicated overhead but is not available at the moment.

ISBN 978-82-14-04396-9

The approach we claim as most useful for the MDPE process is based on Higher-Order Transformations as available and described by Jouault [16] for the Atlas Transformation Language (ATL) [12]. Higher-Order Transformations are transformations that are used to transform one transformation A to a new transformation $A*$. This approach is used in [16] to automatically extend rules within ATL transformations with code for additional information generation. This code creates a tracing model when the transformation is executed. This tracing model conforms to a traceability metamodel, which is defined in [16] by extending the Atlas Model Weaver (AMW) [17] metamodel. This approach is not *traceability by design* as there is the need to consider tracing after implementing a transformation. However, the additional effort is simply executing a Higher-Order Transformation which has only to be defined for each applied transformation language but not for each single transformation. Additionally, the application of the Higher-Order Transformation can be integrated in the transformation tooling.

In our implementation, we execute the transformation provided by [16] to extend our current UML2TIPM transformation with tracing capabilities. Figure 3 depicts how, for instance, the rule "DeviceObject" is extended with traceability model generation capabilities. Hence, if the extended UML2TIPM transformation is executed, not only a TIPM but also a tracing model is generated.



```
rule DeviceObject(
 from
  i: UML!Device
 to
  o: TIPM!ActiveResource(
    name <- 'ActiveResource_'
    +i.name,
    multiplicity <-1,
    schedulingPolicy <-thisModule.
    defaultSchedulingPolicy
    )
}
```

```
rule DeviceObject (
 from
  i : UML!Device
 to
  o : TIPM!ActiveResource (
    name <- 'ActiveResource_' + i.name,
    multiplicity <- 1,
    schedulingPolicy <- thisModule.
    defaultSchedulingPolicy
  ),
  __traceLink : Trace!TraceLink (
    name <- 'DeviceObject',
    sourceElements <- Sequence (__LinkEnd_i),
    targetElements <- Sequence (__LinkEnd_o),
    model <- thisModule.__wmodel
  ),
  __LinkEnd_i : Trace!TraceLinkEnd (
    element <- __elementRef_i
  ),
  ...
```

Fig. 3. Comparison between one ATL rule before (left) and after the HOT (right)

The tracing model enables us to annotate simulation results serialized by the monitor model elements in the TIPMs back to the original development models. Therefore, an Eclipse plug-in has been developed which iterates all monitors of a TIPM. Monitors are associated with Steps and Resources in the TIPM as described in subsection 3.1. It is the purpose of the plug-in to use the generated

ISBN 978-82-14-04396-9

tracing model in order to get the related development model element for the TIPM Step or TIPM Resource, the monitor is attached on.

For instance, if a *LatencyMonitor* is associated with the *Steps* "Initial_Node_1" (as entry) and "Final_Node_1" (as exit), the plug-in would analyze the trace links within the tracing model in order to get a reference to the UML Activity Diagram elements that were sources for the UML2TIPM transformation. In the case of the two *Steps* "Initial_Node_1" and "Final_Node_1" we get references to an *InitalNode* and a *FinalNode* in an UML Activity Diagram.

The actual annotation of the simulation result stored in the *LatencyMonitor*, which has been used as in the example in subsection 3.1, to the development models follows in a second step: The latency (*11.38 ms*) is annotated to the UML *Activity* containing the *InitalNode* and the *FinalNode* to which the trace links point. It has been mentioned that we do not only need to support UML models as development models but also other modeling languages such as proprietary languages used within SAP. A general solution for annotation is not possible since we have to take development language specifics, such as the mechanism used for the actual annotation of simulation results, into account. Therefore, we encapsulated the logic implementing the development language specific annotation of performance analysis results in one module, and the logic implementing the development language independent access of the TIPM and Tracing Model in another module.

In order to realize loose coupling between the modelling language specific part and the modelling language independent part, we used the standard extension mechanism provided by the Eclipse platform.

Thus, we implemented one Eclipse plug-in which implements the development language independent part of the TIPM to development model tracing, and provides an Eclipse Extension Point to be implemented by the development language specific Eclipse plug-in. We implemented such a plug-in in order to annotate the AnyLogic simulation results from the TIPM monitors back to UML models via the SPT profile [18].

By combining our transformation and tracing solutions, we created a prototype, which we successfully applied on examples. An issue often discussed when it comes to applying such transformation chains with tracing is information loss. For the application presented in this paper, however, we did not encounter any issues with transformation loss. Clearly, information is not preserved by our transformations; but that is intended, since only selected information is carried from development to transformation models and a different kind of information—the analysis results—are carried back.

4 Related Work

Our work has been strongly influenced both by needs arising from industrial practice and by previous work in the academic literature. Here, we briefly discuss some of the influences from the literature.

ISBN 978-82-14-04396-9

Grassi and Mirandola with their work in the area of component-based software-performance engineering were the first, in our knowledge, to present the notion of using model transformations in the MDA context for generating analysis models ([19], for example) for analysing performance. They propose refining a second line of models from the most generic models in parallel to those models meant for eventually generating executable code. In contrast, we propose to generate a new analysis model whenever it is needed, basing it on the most current development model available. Other authors have also proposed using model transformations for constructing analysis models. A more detailed discussion can be found in [5].

Our approach is much closer to work performed by Sabetta et al. [20], who present a new technique for transforming development models into analysis models using so-called abstraction-raising transformations. This work could be used as an extension of our work, although we would need to extend their specific transformation technique to support tracing in the way we need it.

Our TIPM metamodel is closely related to Woodside's work on CSM [9]. In fact, the TIPM metamodel is an extended version of CSM. Our main extension is the addition of the concept of monitors that enable us to indicate the specific performance properties of interest. As we have seen, these monitors play an important role in feeding information back into the development model as they will contain the analysis results. Based on the CSM, Woodside has gone on to build PUMA [8], a system quite similar to the work presented here. Feedback to development models is quoted as future work in [8], however.

5 Conclusion

We have presented the implementation of a performance analysis result feedback mechanism for MDPE based on Higher-Order Transformations for ATL. This technique helps developers to understand and experiment with performance effects of design decisions without the need for performance expertise. All that is required is the provision of basic performance annotations in the development models. In cases like the SAP case cited above, even this can be avoided by providing catalogues of available components already pre-annotated with correct performance data.

With the basic MDPE framework in place, we now need to perform experiments to support our claim that the result feedback is actually useful to domain experts. Such experiments will be performed in the experimentation phase of the MODELPLEX project and may lead to corresponding adjustments to MDPE. Also, in this context we will be implementing support for further input languages and simulation engines.

Acknowledgement

This research has been co-funded by the European Commission within the 6th Framework Programme project MODELPLEX contract number 034081 (cf. `http://www.modelplex.org`).

ISBN 978-82-14-04396-9

References

1. Fritzsche, M., Gilani, W., Fritzsche, C., Spence, I., Kilpatrick, P., Brown, J.: Towards utilizing Model-Driven Engineering of Composite Applications for Business Performance Analysis. In: ECMDA-FA'08 (to appear). (2008)
2. SAP AG: SAP NetWeaver Composition Environment 7.1 (2008) https://www.sdn.sap.com/irj/sdn/nw-ce/.
3. D'Ambrogio, A.: A model transformation framework for the automated building of performance models from UML models. In: WOSP '05, ACM (2005) 75–86
4. Gu, G.P., Petriu, D.C.: XSLT transformation from UML models to LQN performance models. In: WOSP '02, ACM (2002) 227–234
5. Fritzsche, M., Johannes, J.: Putting Performance Engineering into Model-Driven Engineering: Model-Driven Performance Engineering. In: MoDELS'2005 Satellite Events: Revised Selected Papers, LNCS 5002, Springer (2007)
6. Franks, R.G.: Performance analysis of distributed server systems. PhD thesis (2000) Adviser-C. Murray Woodside.
7. López-Grao, J.P., Merseguer, J., Campos, J.: From UML activity diagrams to Stochastic Petri nets: application to software performance engineering
8. Woodside, M., Petriu, D.C., Petriu, D.B., Shen, H., Israr, T., Merseguer, J.: Performance by unified model analysis (PUMA). In: WOSP '05, ACM (2005) 1–12
9. Petriu, D.B., Woodside, M.: An intermediate metamodel with scenarios and resources for generating performance models from uml designs. In: Software and Systems Modeling, Volume 6, Issue - 2. (2007) 163–184
10. Knöpfel, A., Gröne, B., Tabeling, P.: Fundamental Modeling Concepts: Effective Communication of IT Systems. John Wiley & Sons (2006)
11. XJ Technologies: AnyLogic — multi-paradigm simulation software (2008) URL http://www.xjtek.com/anylogic/.
12. ATLAS Group: ATLAS transformation language (June 2007) URL http://www.eclipse.org/m2m/atl/.
13. OMG: MOF QVT Final Adopted Specification. (June 2005)
14. SmartQVT: SmartQVT - A QVT implementation (2008) http://smartqvt.elibel.tm.fr/.
15. Kolovos, D.S., Paige, R.F., Polack, F.: The Epsilon Object Language (EOL). In: ECMDA-FA. Volume 4066 of LNCS., Springer (2006) 128–142
16. Jouault, F.: Loosely Coupled Traceability for ATL. In: ECMDA-FA workshop on traceability. (2005)
17. Fabro, M.D.D., Bezivin, J., Jouault, F., Breton, E., Gueltas, G.: AMW: a generic model weaver
18. OMG: UML profile for schedulability, performance, and time specification (January 2005) URL http://www.omg.org/docs/formal/03-09-01.pdf.
19. Grassi, V., Mirandola, R.: A Model-driven Approach to Predictive Non Functional Analysis of Component-based Systems. In: NfC'04 Technical Report TR TUD-FI04-12. (2004)
20. Sabetta, A., Petriu, D.C., Grassi, V., Mirandola, R.: Abstraction-raising Transformation for Generating Analysis Models. In: MoDELS'2005 Satellite Events: Revised Selected Papers, LNCS 3844. 217–226

ISBN 978-82-14-04396-9