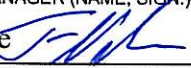

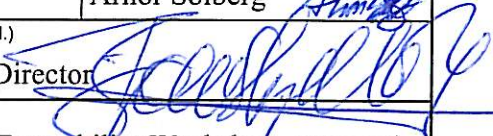


SINTEF REPORT

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		TITLE	
		ECMDA Traceability Workshop Proceedings 2007	
		AUTHOR(S)	
		Jon Oldevik, Gøran K. Olsen, Tor Neple	
		CLIENT(S)	
		ECMDA Traceability Workshop 2007	
REPORT NO.	CLASSIFICATION	CLIENTS REF.	
SINTEF A1378	Open	ECMDA Traceability Workshop 2007	
CLASS. THIS PAGE	ISBN 978-82-14-04056-2	PROJECT NO.	NO. OF PAGES/APPENDICES
		90B234	68 / 0
ELECTRONIC FILE CODE		PROJECT MANAGER (NAME, SIGN.)	CHECKED BY (NAME, SIGN.)
ECMDA-TW Proceedings		Tor Neple 	Arnor Solberg 
FILE CODE	DATE	APPROVED BY (NAME, POSITION, SIGN.)	
	2007-06-06	Bjørn Skjellaug, Research Director 	
ABSTRACT This report contains the proceedings of the Third ECMDA Traceability Workshop, arranged in Haifa, Israel 2007 together with ECMDA 2007 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



MODELPLEX

ECMDA Traceability Workshop (ECMDA-TW) 2007 Proceedings

June 12th 2007, Haifa, Israel

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



ISBN 978-82-14-04056-2

Table of contents

Introduction	5
Improving Traceability in Model-Driven Development of Business Applications (<i>Rummeler, Grammel, Pohl</i>)	7-15
RT-MDD Framework - A Practical Approach (<i>Costa, da Silva</i>)	17-26
Model-based Methodology for Requirements Traceability in Embedded Systems (<i>Albinet, Boulanger, Dubois, Peraldi-Frati, Sorel, Van</i>)	27-36
Traceability as Input for Model Transformations (<i>Vanhooff, Van Baelen, Joosen, Berbers</i>)	37-46
Traceability and Provenance Issues in Global Model Management (<i>Barbero, Del-Fabro, Bézivin</i>)	47-55
Traceability-based Change Management in Operational Mappings (<i>Kurtev, Dee, Goknil, van den Berg</i>)	57-67

10

Introduction

This is the third Traceability Workshop organised in the context of the ECMDA conference series. The two previous workshops have focused mainly on theoretical and technical challenges of traceability in model-driven engineering. In this year's workshop, our goal is to put more focus on practical applications of traceability. This is reflected in the workshop papers that represent a mixture of practical and theoretical work.

We hope that the workshop papers act as catalysts for fruitful discussions both on practical and theoretical solutions and challenges in model-driven traceability. What are the current showstoppers for adopting traceability with respect to tools, processes, languages, and mechanisms? This is one question we aim to get some insights on during the 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 2007.

[†] 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.

10

Improving Traceability in Model-Driven Development of Business Applications

Andreas Rummler¹, Birgit Grammel¹ and Christoph Pohl²

¹ SAP Research CEC Dresden,
Chemnitzer Str. 48, D-01187 Dresden, Germany

² SAP Research CEC Karlsruhe,
Vincenz-Priessnitz-Str. 1, D-76131 Karlsruhe, Germany

E-Mail: {andreas.rummler,birgit.grammel,christoph.pohl}@sap.com

WWW: <http://www.sap.com/research>

Abstract. This paper gives an overview of the support for traceability in an industrial context. The state of best practices are described and shortcomings are identified. Furthermore the AMPLE project is introduced, encompassing an overview on our approach towards traceability in the context of Model-Driven Development (MDD) and Software Product Lines (SPL) in conjunction with Aspect-Oriented Software Development (AOSD). AMPLE can be considered as a joint effort to tackle the shortcomings of current industrial practice. The paper is of positional nature and outlines our work currently in progress.

1 Introduction

During the development stages of complex software systems many artefacts are generated, either manually or in an automatic manner. The nature of these artefacts ranges from requirements expressed in text documents down to statements in source code. Understanding the mutual dependencies and the logical relationships among artefacts is a nontrivial task and becomes harder the more complex a system is. The problem is not only of academic interest - it has also been perceived in industry. According to an internal audit of customers of SAP, missing traceability during the whole development cycle is the top-rated weakness.³ In addition, missing traceability information was explicitly mentioned as weakness in 2005 in an external ISO certification audit.

In real world business applications traditional model-driven development approaches and software product line engineering techniques often do not reflect the decomposition of system features well enough. For instance, compliance checks of legal business regulations or late introduction of security properties often crosscut the architectural design of a system. To overcome these issues Aspect-Oriented Software Development (AOSD) technologies are gradually taking hold in industrial software development [1, 2].

³ In 2004, 10 out of 11 customers criticized this.

AOSD techniques aim to modularize crosscutting concerns, that are scattered over the whole system into independent modules. These modules are called aspects. There is no clear mapping of aspects identified during the requirements stage to later development stages. This is due to the fact, that aspects have complex dependency relations, i.e. aspects are related to other artefacts within one or more phases. Secondly tracing of aspects is aggravated through their versatile nature, e.g. appearing or disappearing within a phase. This results in increased complexity for the task of providing traceability in systems utilizing AOSD techniques [3]. Model and code weaving techniques make decisions that a tracing mechanism needs to take care of and needs to keep track of.

While traceability in MDD seems to be relatively well understood, traceability in conjunction with AOSD techniques is a rather new field of interest. In MDD several approaches have been implemented and are used in practice. To mention one, traceability is supported in Query View Transformation (QVT), [4, 5], where instances of trace classes store the record of transformations. However, in these approaches, traceability starts in the design stage. For this reason tracing of artefacts created in earlier stages like requirements engineering to later development stages is still poorly understood.

The remainder of this paper is structured as follows. First, we will give an overview on the traceability support, which is currently implemented at SAP. In section 3, we then continue to introduce the AMPLE project and state the main objectives. In section 4, the focus is shifted to potential means and ways of improving traceability in the context of AMPLE. Last, but not least a few concluding remarks and acknowledgements are given in section 5 and 6 respectively.

2 Traceability Support inside SAP

The product development process inside SAP is backed up by an integrated process model called *Product Innovation Lifecycle* (in the following abbreviated as *PIL*). PIL subdivides the whole product lifecycle into several stages, reaching from the initial idea to deployment and maintenance. The individual stages are *Invent*, *Define*, *Develop*, *Deploy* and *Optimize* and are shown in figure 1. Although not explicitly shown in the mentioned figure, PIL is an iterative process, i.e., phases may be passed several times.

The *Invent*-stage is the initial stage. Here ideas that could be interesting to a specific market are collected, evaluated and categorized. Typical activities during this stage incorporate the definition of market and solution requirements and the analysis of how a potential product idea fits into the overall corporate and solution strategies. The requirements originating from this stage are transferred to development actions in the succeeding *Define*-stage and realized in concrete development projects in the third stage, the *Develop*-stage. This stage incorporates all typical activities of a development process: From architecture definition and high level planning over detailed design to implementation, integration and unit/acceptance test. The last two stages include the actual roll-out

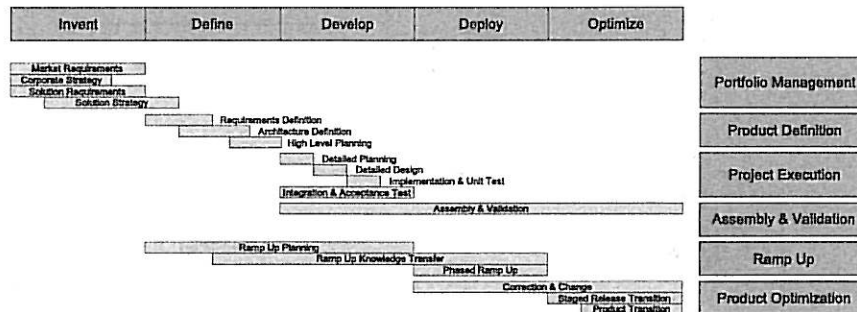


Fig. 1. Product Innovation Lifecycle Outline

to the customer and a continuous on site optimization, including corrections and late changes.

PIL is supported by several tools, which as such are only partially available as commercial products. There is tool support assigned to the planning stages; a user is aided in his tasks in product portfolio management and product definition. This covers the stages of *Invent* and *Define*. Here a coarse-grained view on a certain product or product family is facilitated. Market requirements can be grouped together into so-called portfolio cases and assigned to work packages.

More interesting in this context is the tool support for the next stage. This incorporates project planning, execution, accounting, resource and time management. The already defined work packages are assigned to newly created projects which can be linked to actual software requirements for the purpose of providing traceability. The idea behind this is to allow tracing of information from the definition of a product to its delivery; requirements are meant to be arranged in a way to enable their own management and the management of their interdependencies. artefacts that are created to fulfill requirements are arranged in a continuous path. At certain points along this path deliverables are provided; the contained information inside is reused later on. The requirements themselves serve as anchors for traceable paths. To enable this, requirements must be structured into coherent pieces, which can be interlinked. Such a formalized outline reduces the danger of misinterpretation, which, of course, cannot be fully eliminated.

The tool support provides templates for the creation of new requirement specification documents. These documents are structured into a header, several chapters and subchapters. The chapters contain requirements tables, that allow the definition and linking of single requirements, the definition of identifiers, types, trace-to statements and appropriate scope. These documents are finally linked to projects where the actual activities are performed to fulfill the requirements. On the back-end a relation engine allows the linking of requirements to test cases to ensure that a certain requirement is covered by one or more tests in the final product.

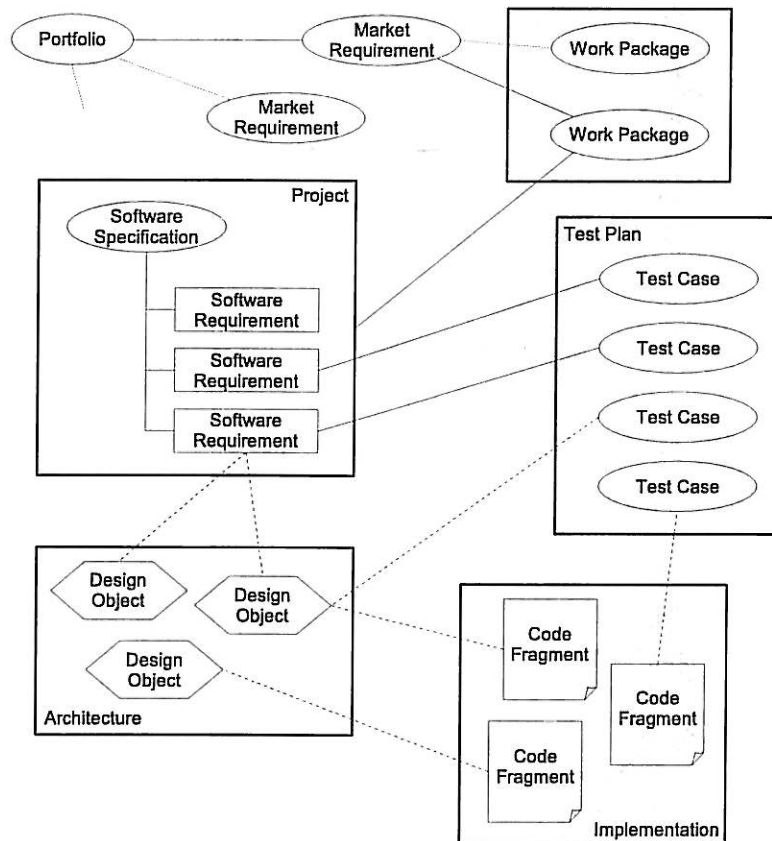


Fig. 2. Linking of artefacts in the Development Process

The result of each stage is a set of particular deliverables. Documenting the transition between single deliverables could be done in a graphical way supported by dedicated tools. At the moment this is not implemented, instead standard office tools are used for these tasks. Examples are the manual creation of text or spreadsheet documents. Some steps in this process are performed in a model-based way, but are in fact still far away from a truly model-driven approach.

Looking at the full traceability picture (shown in figure 2) of the entire development process, which is currently implemented, this incorporates the linking of market requirements to software requirements specifications documents which in turn are linked to their entailing software requirements. These software requirements then can be linked to test cases. Further linking of requirements to design and/or development artefacts, respectively the linking of those artefacts to test cases is possible in general, but not sufficiently supported by tools. This

is indicated in figure 2 by dashed lines and is subject to future work. In the current state several interesting questions that arise during the development process cannot be answered in an automatic and reasonable way:

- How can it be ensured, that the design which was produced by developers really covers all requirements?
- Are the market requirements, which were defined on a very high and abstract level, really implemented and how can this implementation be tested against these requirements?
- Have artefacts been developed in one or more stages, such that they cause a behaviour which is not covered by the requirements specification or even is unwanted?

These questions can be answered on a fine-grained level by experts involved in the development process, but not by people dealing with a coarse-grained view. Therefore, tools are needed that are able to give reasonable answers to these questions or at least help people in finding those answers. The current tool support inside PIL is based on conventional development methodologies, which do not cover new approaches like AOSD but may help in easing the task of tracing artefacts. This is where the AMPLE project comes into play, which is described in more detail in the next section.

3 The AMPLE Project

AMPLE stands for Aspect-Oriented, Model-Driven Product Line Engineering⁴. The project is funded by the European Union and incorporates nine project partners, being from the field of academic science as well as industry, the latter including SAP. The overall aim of the project is to develop a methodology for the design of Software Product Lines (SPLs) that offers improved modularisation of variations to support their traceability across the entire SPL lifecycle. To achieve this a novel combination of AOSD and MDD techniques is applied. The focus is on providing a holistic treatment of variability by not only addressing variability at each stage in the SPL life cycle but also managing variations in associated artefacts such as requirements documentation, manuals and reports. Furthermore, AMPLE aims to bind the variation points in various development stages and dimensions into a coherent variability framework across the SPL engineering life cycle thus providing effective forward and backward traceability of variations and their impact. This makes it possible to develop resilient yet adaptable software product line architectures for exploitation in industrial SPL engineering processes.

The objectives of AMPLE cover all stages of the development process from the early stages of requirements analysis and engineering over design to implementation. Most interesting in this context is a separate work package with the goal to develop aspect-oriented models representing the interfaces between requirement analysis-design, design-implementation and implementation-execution

⁴ The website for the project can be found at <http://www.ample-project.net>

phases in order to provide an abstract view of the refinement of the software product line in order to maintain forward and backward traceability of the variations. Along the associated infrastructure and methodologies for software product line engineering for smooth transition to industrial software product line engineering processes should be developed.

4 Improving Traceability in AMPLE

Within the AMPLE project SAP is involved – to a large degree – in research on traceability. Questions originating from industry practice should be investigated further to come to practicable solutions. Besides the general goals in AMPLE, namely improving transition from model-based to model-driven development and the modularization of cross-cutting concerns into reuseable aspects, SAP is especially interested in synergies arising from the combination of SPL engineering approaches in existing SAP solutions with model-driven approaches and of course in the traceability of all modelled artefacts across the whole software lifecycle. Although traceability has been recognized as an important issue in SPL engineering, reliable and industrial-strength tool support is missing, from the commercial side as well as in form of academic prototypes.

Traceability refers to the linking of different artefacts on the same and on different abstraction levels and the ability to follow those links. artefacts may include documents, stakeholders, modelled objects or code fragments. It can be easily noticed, that these vary in nature and structure. artefacts may be divided into different kinds of subgroups: As an example, documents may be classified as requirements, within a glossary, use cases or source files. In addition, links between artefacts may also be of different kinds, the semantic meaning of links may include *refine*, *implement* or *substitute*.

A recent survey of general approaches on traceability done in the AMPLE project found several open problems and shortcomings in existing techniques:

- SPL engineering approaches are not capable of providing traceability between artefacts on different abstraction levels, i.e. between market requirements and software requirements [6], [7], [8].
- Although linking between software requirements and test cases is already applied in industry, the applied techniques are usually tailored to special cases and are not applicable within a general context.
- Tracing information about product derivation is a complicated task. This information may include configuration data, build information or version numbers, which is scattered over project files with proprietary data formats. In addition available tool support in general is poor.
- There is no integrated connection between traceability tooling and software configuration management systems.
- Change impact and coverage information is not available end-to-end for crosscutting concerns, subtle changes at both ends of the development stages may impact artefacts in the whole product line [9], [10].

- It is impossible to analyse generated artefacts for potentially unwanted side-effects.

These problems are to be addressed in the AMPLE project. There are approaches to Aspect-Oriented Requirements Engineering (AORE) [11, 12] that can help address some of these shortcomings. For instance relationships, dependencies and interactions among existing requirements can be identified at early stages of the development lifecycle. However, AORE approaches do not explicitly define mechanisms for mapping information gathered at the requirements level to later development phases. There is a need for defining mapping guidelines, rules, and heuristics for mapping of entities and trace information across the entire development lifecycle. Asset repositories are also required that may collect and maintain product line assets and the mapping rules, guidelines, and heuristics. In addition, there is a need for a traceability meta-model defining which trace information, i.e., assets, concerns, relationships, dependencies, behaviours, compositions and mappings, needs to be captured and managed.

The approach followed in the AMPLE project relies on the modularization of cross cutting concerns at model level. Starting already at the stage of requirements engineering will foster traceability. To be able to track dependencies between AO and non-AO artefacts along the development cycle, defining explicit aspect interfaces seems advantageous. Earlier work on aspect-aware interfaces [13] and Extension Join Points [14] confirms this idea. The intrusive nature of AO techniques is reduced in its intensity by defining aspect interfaces that partly abandon the obliviousness property but make the contract between the to-be-extended system and the extending aspects more explicit. Although this primarily improves maintainability, it also enhances traceability. The challenge is to apply this concept not only at code level but also in design models or even in requirements engineering.

First experiences in combining aspect-oriented principles with model-driven software development yielded promising results. This work was based on open-ArchitectureWare [15, 16]. Work in progress will leverage on these experiments on aspect interfaces, extend this approach and incorporate support for tracing.

Ongoing work consists of the definition of a metamodel for variability in SPL including the support of AO concepts and appropriate tracing information. Based on this metamodel a tool chain is designed that supports the definition of SPL, product generation and full support for tracing relationships and dependencies among automatically generated or manually created artefacts. A suitable use case that is currently implemented consist of a complete example originating from SAP's core business. Here, the ideas and concepts elaborated in the AMPLE project are examined and evaluated from a very practical perspective.

5 Conclusion

Enabling traceability of artefacts throughout all stages of the development cycle of a software system has been recognized as a crucial task. But there is a large gap between needs and best practices in industry on one side and published solutions

from academic research on the other. This article gives an introduction to work related to traceability in the AMPLE project, which intends to solve some of the problems related to this field of research. Inside AMPLE, SAP follows an approach where explicit extension points are defined that can be complemented by appropriate aspects to moderate the highly intrusive nature of AO techniques. Following such an approach will remove the 'obliviousness' of aspect orientation up to a certain degree, but is to the authors' opinion the only way to achieve greater acceptance of these techniques in an industrial context. A developer is now forced to take care of potential aspect-oriented extensions to the code he is creating. In addition this concept should not only be utilized in implementation but also raised to the levels of modelling or even requirements engineering. This will increase maintainability and along with that traceability in general, but especially vertical traceability among artefacts on the same abstraction level. Based on this approach a tool chain is developed that includes complete support of traceability from one end of the development process to the other.

6 Acknowledgements

This work has been partially funded by the European Union in project AMPLE, FP6 IST SO 2.5.5.

References

1. Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.M., Lopes, C.V., Maeda, C., Mendhekar, A.: Aspect-oriented programming. In: Proceedings of European Conference of Object-Oriented Programming ECOOP 97. (1997)
2. Filman, R.E., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development. Addison-Wesley (2005)
3. Katz, S., Rashid, A.: From aspectual requirements to proof obligations for aspect-oriented systems. In: Proceedings of the 12th IEEE International Conference on Requirements Engineering. (2004)
4. Helsen, S.: Model Transformations with QVT. In: Model Driven Software Development. John Wiley & Sons (2006) 203 – 222
5. Object Management Group: MOF 2.0 Query View Transformation (ad/2005-03-02). (2005)
6. Rashid, A., Moreira, A., Araujo, J.: Modularisation and composition of aspectual requirements. In: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development. (2003)
7. Moreira, A., Rashid, A., Araujo, J.: Multi-dimensional separation of concerns in requirements engineering. In: Proceedings of the International Conference on Requirements Engineering. (2005)
8. Ruzanna Chitchyan, e.: Semantics-based composition for aspect oriented requirements engineering. In: Proceedings of the Sixth International Conference on Aspect-Oriented Software Development. (2007)
9. Yu, Y., d. P. Leite, J.C.S., Mylopoulos, J.: From goals to aspects: Discovering aspects from requirements goal models. In: Proceedings of the 12th IEEE International Requirements Engineering Conference. (2003)

10. Cleland-Huang, J., Settimi, R., BenKhadra, O., Berezhanskaya, E., Christina, S.: Goal-centric traceability for managing non-functional requirements. In: Proceedings of the 27th International Conference on Software Engineering ICSE 05. (2005)
11. Grundy, J.: Aspect-oriented requirements engineering for component-based software systems. In: Proceedings of the 4th IEEE Symposium on Requirements Engineering. (1999)
12. Tekinerdogan, B., Moreira, A., Araujo, J., Clements, P.: Early aspects: Aspect-oriented requirements engineering and architecture design. In: Workshop Proceedings at AOSD Conference. (2005)
13. Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: Proceedings of the ACM International Conference on Software Engineering. (2005)
14. Kulesza, U., Alves, V., Garcia, A., Lucena, C., Borba, P.: Improving extensibility of object-oriented frameworks with aspect-oriented programming. In: Proceedings of the International Conference on Software Reuse ICSR 06. (2006)
15. Völter, M., Kolb, B.: OpenArchitectureWare und Eclipse. Eclipse Magazin (7) (2005)
16. Völter, M.: MDSD und/oder AOSD? Java Spektrum (2) (2005)

10

RT-MDD Framework – A Practical Approach

Marco Costa¹, Alberto Rodrigues da Silva²

¹ Univ. Lusíada
Rua da Junqueira, 188-198
1349-001 Lisboa, Portugal
mbcc@acm.org

² INESC-ID/Instituto Superior Técnico
Rua Alves Redol, Nº 9
1000-029 Lisboa, Portugal
alberto.silva@acm.org

Abstract. Traceability in software engineering has been handled mostly as a documentation activity. This paper describes a reactive approach to traceability using MDD as general orientation and QVT as a transformation language. A conceptual model is presented defining some of the key concepts of a reactive traceability framework. Also the semantics of these key concepts is explained. Some examples are given of the framework's artefacts.

Keywords: Reactive traceability, MDD, QVT, UML, RT-MDD

1 Introduction

An information system involves a set of active artefacts that cooperate towards a common goal. These artefacts are present in multiple views, regarding the abstract layer we take as a viewpoint. For instance, a system may be described by its functions, data structures and technologies, among other features. Documentation is a part of the solution, just as the formulation of a problem may be considered part of its solution.

Nowadays information systems are growing in complexity, i.e., in terms of the number of artefacts and relations between them. Development of new applications and maintenance of the existing ones should be accompanied by tools and methodologies which minimize the risk of introducing a state of incoherence between the artefacts. When this problem is not tackled the reality shows that programming code evolves with no or minor relation with models. In large applications, when models are almost completely outdated the system is in risk of becoming unmaintainable.

Traceability, is used in many different contexts from food industry to software development and maintenance. We focused our work in traceability between artefacts of models and code (Fig. 1). The requirements traceability [11] was left outside the

solution for now, but this issue may be integrated in the future work, as other types of traceability.

Traceability deals with keeping records of relations between artefacts of the same, or different abstract levels. We propose the term *reactive traceability* as a characteristic of a system that not only keeps information of the relations, but also can react to and prevent changes to artefacts or its relations (traces). The goal of the RT-MDD (Reactive Traceability Model Driven Development) framework is to maintain a state of coherence between the artefacts of the system.

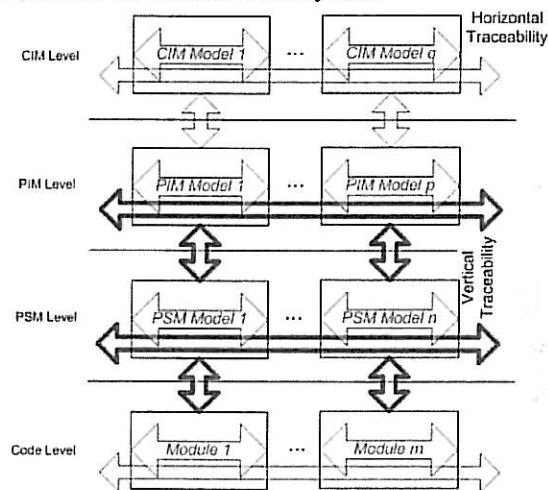


Fig. 1. Traceability and MDA (bolder arrows defines focus of this paper).

Automated transformation of models-to-models, models-to-code, as well as code-to-models, is becoming a reality [19, 20, 21]. These transformations generate output artefacts from source artefacts. The relation between input and output artefacts of a transformation is just one, and obvious, type of semantic relations among others. Object Management Group QVT (Queries, Views and Transformations) [3] is aimed to standardize not only transformations with models but also other operations with models, like queries and views. Our approach takes QVT as a starting point to accomplish the construction of automated transformations between models and implements a way of maintaining traces between artefacts (models and code) as well as reacting to changes for the sake of system coherence.

Our proposal is aligned with OMG standards and recommendations like UML [5] and QVT [3]. Even if QVT by itself has not a complete implementation, it is however a starting point to different approaches and products like Tata ModelMorf [10] or Compuware OptimalJ [7]. Other concurrent approaches to QVT include ATL [8], Mistral [6] and EML [18].

Gorp's [12] *consistency* is a similar concept to coherence in this paper. The vision of traceability in a MDD perspective is shared with other initiatives, such as those in [14, 15, 16].

The paper is structured as follows. Section 2 proposes the conceptual model of a traceability solution and defines traceability from a systemic approach. Section 3 identifies different types of dependency relations between artefacts. Section 4 relates transformations with traceability and explains how QVT define traces. Section 5 describes some important features of the traceability solution prototype implemented.

2 Artefacts and Traceability

The relevant information for traceability (Fig. 2) corresponds to artefacts, which are products generated or crafted by tools used in a development process. In this context an artefact may be any piece of code, or model object, with relevant properties regarding traceability; for instance, an UML class, a C# method, or a SQL Server table definition.

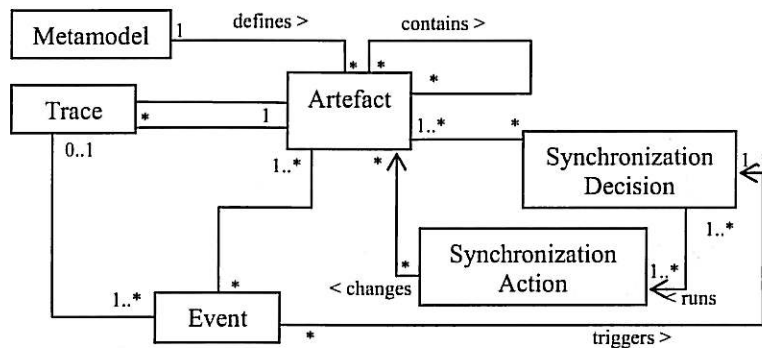


Fig. 2. Conceptual model of the reactive traceability framework.

The artefact can be any object used, modified or made in the context of the development environment, included in the system, and accessible by the reactive traceability solution. Artefacts should be defined through of a metamodel.

For a programming language to be included in this solution it must have a metamodel, even if not covering all possible concepts, if irrelevant to the level of traceability needed. Our approach uses MOF to express metamodels. Artefacts may be transformed in other artefacts, using some kind of transformation process available (fully automatic, assisted or manual) and traces are maintained when a transformation occurs. Of course, transformation is just one of different possible types of semantic relations between artefacts. Considering legacy systems, a transformation may not be possible if the code cannot be changed. In cases like this, maybe it is necessary to record manually different relations expressed (e.g, considering a table, named *client*, in the legacy system related to a UML class, named *entity* in a class diagram). As a consequence, there is a need of other types of dependency relations, other than transformations.

3 Dependency Relations and Traces

Different dependency relations are possible between artefacts. We can identify two types of classification: a) *conceptual level* and b) *semantic coupling*. In a) we can consider artefacts in different conceptual levels [4] (vertical traceability), in the same level (horizontal traceability) and within the same model or in different ones (intra and inter-model traceability) as seen in Fig. 1.

Each dependency relation can also be characterized by the level of coupling that exists between the artefacts. Let us define the *equivalence* relation between two artefacts of a model, or from different models, as when they have the *same representation of the same concept*. In this context *same representation* stands for having the same data that identifies the concept, even if the artefacts are in different levels of abstraction, or artefacts.

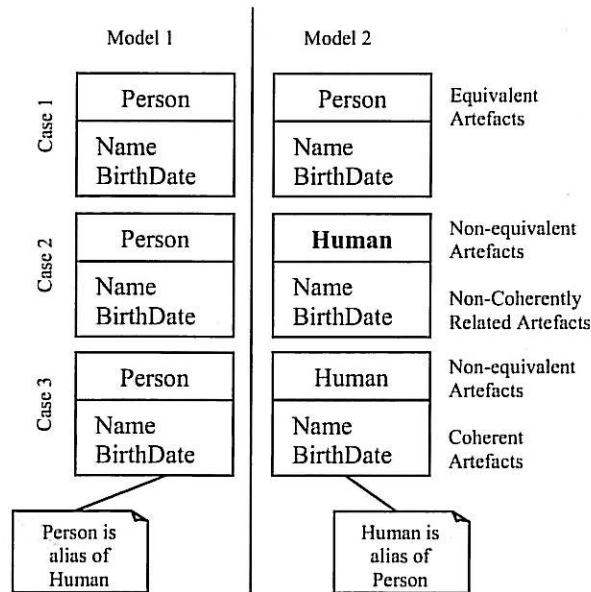


Fig. 3. Dependency relations between models.

Considering Fig. 3, there are three cases which illustrate differences between dependency relations. In *Case 1* the two artefacts are **equivalent**, as they have the same representation of the concept *Person*, considering different models. In *Case 2* the classes have different names so they are not equivalent. In the same case we can't say if they are coherent or not, as there is no information in the models relating the two classes. In *Case 3* the two concepts are **coherent**, with added simplified information to each class, as there is a semantic relation between the two artefacts. The semantic relation is not always trivial to observe, considering complex legacy applications, different natural language names to concepts, or even different programming and design paradigms. In real work coherence is more difficult to

maintain than the other two relations. It is however the kind of relation that is necessary to deal with if reactive traceability is to be maintained between the artefacts. Each change to an artefact may trigger an event which may change or not the coherence state of the system itself. If a part of a system is in an incoherent state, the system may be in risk. In consequence, the state of the system must be checked when a change of an artefact occurs and a decision may be required to run existing actions that will bring again the system to a coherent state.

In this perspective a trace is seen as a record of a dependency relation between two artefacts. The artefacts may have different levels of abstraction, e.g., it is possible to define a trace between all classes of an UML class model and a set of table definitions. The artefacts can be at a metalevel, i.e., at a language definition level (e.g. artefact *Class* of metamodel *UML*). As the artefacts also include structural artefacts (e.g. the UML class *X*) generic rules of traceability may not hold for all necessary traces. A reactive traceability solution must maintain the traces and ensure that they are still valid at any time.

4 Reactive Traceability and Transformations

Transformations between models or between models and code (usually *from* models *to* code and not the opposite) are a relevant issue to reactive traceability. After transformations are performed traces are created (implicitly or explicitly) but this is not the only action that can create traces. There are two viewpoints (Fig. 4) to the issue of creating traces: a) in legacy applications one may consider artefacts already existent and traces will instantiate and describe some implicit or explicit semantic relations, or dependencies, between them; b) from a starting point in an artefact (diagram or code) it is necessary to create (or generate) another artefact, using some type of transformation.. These two viewpoints are both necessary in a solution that implements reactive traceability. The first item has a focus on creating and maintaining the semantic relations between existing artefacts and the word *existing* is a keyword to understand this viewpoint. At some time of the development process changes in the system will become evolutions of an existing state. For each change the system coherence is checked and decisions are made about the new achieved system state. The second gives more importance to the generation process [1, 2] and traces are relations between *old* and *new* (generated) artefacts. The second viewpoint is related to the IEEE traceability definition in [13]: *"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"*. When a generation of a set of artefacts occurs it is necessary to record the new dependency relations that are created in the process. These two approaches are not only valid but necessary, in a development process and reactive traceability must include both.

Transformations between models, and by extension between models and code, may be realized using a standard like QVT [3]. The QVT specification uses trace classes to record traceability data between artefacts. QVT has a declarative language (Relations Language – QVT-RL), an imperative language (Operational Mappings Language –

QVT-OML) and a simplified core language (QVT-Core) in which the other more complex language constructs may be expressed.

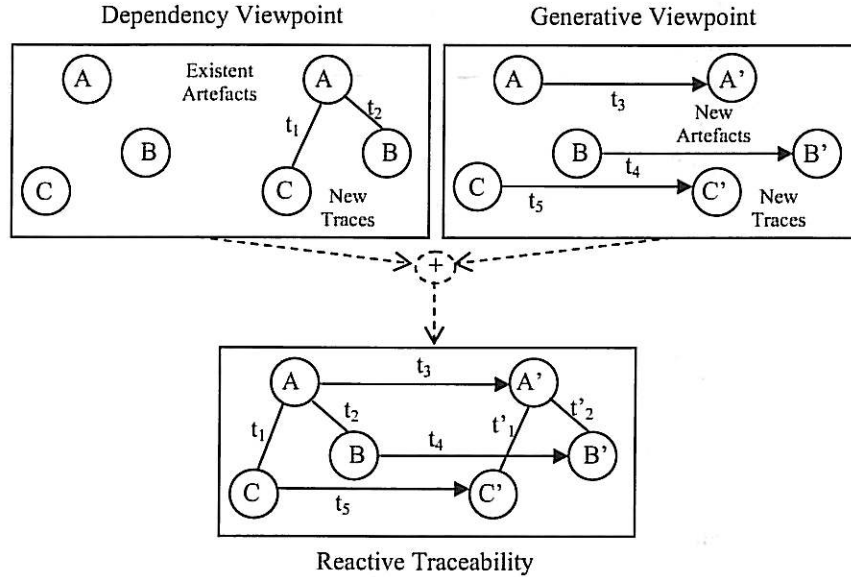


Fig. 4. Creation of traces: two approaches to reach reactive traceability.

In the QVT-RL the traces are implicitly defined (which, in this context, is almost the same as undefined), but there is a trace class generation rule that defines the corresponding trace class definition in the QVT-Core language. The generation rule states that [3] "Corresponding to each relation there exists a trace class in core. The trace class contains a property corresponding to each object node in the pattern of each domain in the relation". Let us consider a simplified QVT relation between a class diagram at design level written in UML and the corresponding C# classes:

```

relation UMLClassToCSharpClass
{
  checkonly domain uml uc: UClass {
    namespace = un:UMLNamespace {},
    name = cn }
  checkonly domain csharp csc: CSharpClass {
    namespace = csn:CSNamespace {},
    name = cn }
}

```

The generated QVT-Core trace class is:

```

class TUMLClassToCSharpClass
{
  uc: UClass;
  un: UMLNamespace;
}

```

```

    csc: CSClass;
    csn: CSNamespace;
}

```

When instantiated, this trace class will represent the actual dependency relation between the two related artefacts.

5 RT-MDD Framework

As seen later, transformations are an important source of traceability information but are not the only one available. Different tools can create artefacts that may include other artefacts with traces that must be tracked and maintained. Considering transformations as a necessary tool, implemented by a *QVT Engine*, to efficiently produce models and code artefacts, Fig. 5 shows the high-level architecture of a reactive traceability solution. The RT-MDD framework has *providers* which are integration components to different tools. Of course, more components from different types may be added if they produce or consume artefacts with traces. The link between the Traceability Engine and each other component varies from type of provider. Each tool to be included in the traceability solution must have the necessary *provider plug-in* that will be used to access the relevant artefacts which have traces. The relations between artefacts are kept within the RT-MDD solution.

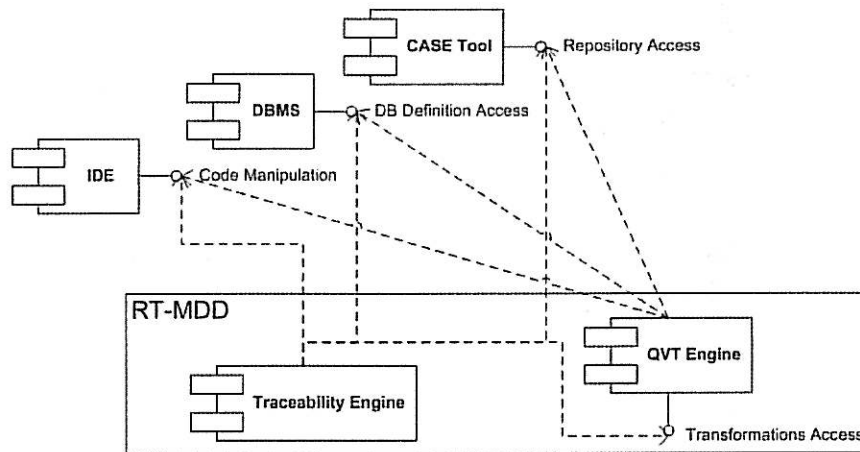


Fig. 5. UML component diagram of a traceability solution.

It is important to note, however that the traceability solution is not a production or test environment tool. It is designed to be used in development environments. As it acts with the structure of the target applications, if the structure is not changed there is no need to control traceability.

The Traceability Engine (TE) permanently verifies the coherence state of the system in a three phase approach: (1) polling, (2) artefacts change handling, and (3) trace change handling.

(1) The TE polls each artefact. When a change is made to an artefact, the TE adds that artefact to a *list of changed artefacts*.

(2) For each artefact in the *list of changed artefacts* TE searches in an *artefact event list* all the relevant events related to that artefact. For each of these events found, it is necessary to verify if a valid action was made in the system. Possible valid actions are: *create*, *update* and *delete*. The *read* action is not valid as it does not change the system state. The *create* action is valid when used with metamodel artefacts. Only then it is possible to say if an event related directly to the artefact exists. If this is the case, the event is triggered. This phase ensures that actions over artefacts and their properties don't change the coherence state of the system.

(3) After artefact events are verified it is necessary to check if traces still hold (even after eventual artefact changes). For each artefact in the *list of changed artefacts* TE recursively searches trace events that refer artefacts contained in the artefact. If a trace event is found, and the relevant artefact and action hold, the event is triggered.

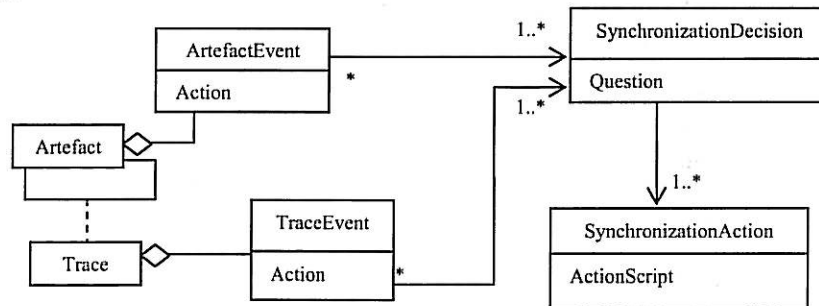


Fig. 6. Domain model of the trace engine

When the trigger conditions are satisfied TE runs synchronization decisions (Fig. 6). When the user selects one of the decisions available TE runs a set of synchronization actions associated to that decision. After the decision is taken the system is in equal or more coherent state than before. If a decision of solving a coherence issue is postponed, a warning is generated and logged in a *to-do list* for further processing.

The user interaction with the solution is minimized if trace and artefact events have only one synchronization decision (representing just one option). In that case TE may enforce or omit the related actions (in this case generating an entry in the *to-do list*).

A prototype is currently under development which has a graphical user interface with QVT diagrams to define transformations and templates.

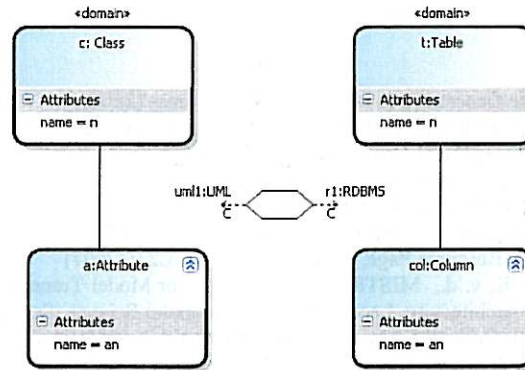


Fig. 7. A QVT Relations diagram (made with the prototype application)

6 Conclusions

Traceability is still regarded as a documentation activity. If the human resources involved in the project are not conscientious about the relevance of documentation in maintenance phases, traceability is not seen as a critical issue. In the software engineering area, traceability has to address specific issues, such as documentation, model production and maintenance, model and code transformations and artefact coherence. This work presents some issues that were considered in the production of RT-MDD framework.

From the early 1970s there is work on requirements traceability, usually with a predominant human operation. Nowadays, the presence of traceability can be very important for the acceptance of MDD and Model Driven Architecture (MDA) in an industry environment. Without tools supporting traceability, across all system levels, the traditional resistance [9, 17] in MDD acceptance will continue to exist.

In our perspective, traceability is not about documentation of the system, it should be more focused on the system itself, as well as its parts and the way they are related to each other. Reactive traceability is a proposal with that orientation. A *reactive* solution is supposed to act, creating, updating, deleting or modifying artefacts of the target system. The number and importance of available artefact types and the level at which the tool is able to interact with each of them is an important measure of its capabilities.

Traceability is already present in many tools available, such as IDEs, CASE tools, RDBMS, requirements tools. The way each tool solves the traceability problem within the context of its artefacts is usually a black-box approach. By definition, traceability uses artefacts from several providers, each one with a possible different metamodel. Metamodeling and transformations are the key tools that traceability uses to provide effective coherence between artefacts of a system.

References

1. Dollard, K.: Code Generation In Microsoft .NET. Apress (2004)
2. Herrington, J.: Code Generation In Action, Manning Pub. Co (2003)
3. OMG: Object Management Group, MOF QVT Final Adopted Specification, www.omg.org/docs/ptc/05-11-01.pdf, (2005)
4. Costa, M., Silva, A. R. d.: Synchronization Issues in UML Models. International Conference on Enterprise Information Systems, Funchal - Portugal (2007)
5. OMG (ed.): UML Resource Page, www.uml.org/#UML2.0 (2007)
6. Kurtev, I., Berg, K. v. d., MISTRAL: A Language for Model Transformations in the MOF Meta-modeling Architecture, Lecture Notes in Computer Science, Springer (2005)
7. Compuware (ed.): Optimal J, www.compuware.com/products/optimalj/ (2007)
8. Eclipse.org (ed.): ATL, www.eclipse.org/m2m/atl/ (2007)
9. Iivari, J.: Why Are CASE Tools Not Used, in Communications of the ACM, Oct.1996, Vol.39, Nr.10, Pgs. 94-103, Association for Computing Machinery (1996)
10. Tata (ed.): ModelMorf – A Model Transformer, www.tcs-trddc.com/ModelMorf/index.htm (2007)
11. Palo, M.: Requirements Traceability. Seminar Report, Department of Computer Science. University of Helsinki (2003)
12. Gorp, P. V., Altheide, F., Janssens, D.: Traceability and Fine-Grained Constraints in Interactive Inconsistency Management. 3rd ECMDA Traceability Workshop (2006)
13. IEEE (ed.). IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. Institute of Electrical and Electronics Engineers (1990)
14. Walderhaug, S., Johansen, U., Stav, E., Aagedal, J.: Towards a Generic Solution for Traceability in MDD. 3rd ECMDA Traceability Workshop (2006)
15. Aizenbud-Reshef, N., Nolan, B. T., Rubin, J., Shaham-Gafni, Y.: Model Traceability. IBM Systems Journal, Vol. 45, Nr. 3 (2006)
16. Oldevik, J., Neple, T.: Traceability in Model to Text Transformations. 3rd ECMDA Traceability Workshop (2006)
17. Welsh, T.: How Software Modelling Tools Are Being Used. Enterprise Architecture Advisory Service Executive Update Vol. 6 , N. 9, 2003-12, Cutter Consortium (2003)
18. Kolovos, D.S., Paige, R. F., Polack, F. A. C.: On-Demand Merging of Traceability Links with Models. 3rd ECMDA Traceability Workshop (2006)
19. Bézivin, J., Lemesle, R.: The sBrowser: a prototype Meta-Browser for Model Engineering. OOPSLA'98 Workshop on Model Engineering, Methods and Tools Integration with CDIF, Vancouver (1998)
20. Akehurst, D.: Proposal for a Model Driven Approach to Creating a Tool to Support the RM-ODP, The 8th International IEEE Enterprise Distributed Object Computing Conference, Monterey, USA, (2004)
21. Perini, A., Susi, A.: Automating Model Transformations in Agent-Oriented Modelling, Proceedings of 6th International Workshop AOSE 2005, Utrecht (2005)

Model-based Methodology for Requirements Traceability in Embedded Systems

A. Albinet¹, J-L. Boulanger², H. Dubois³, M-A. Peraldi-Frati⁴, Y. Sorel⁵, and Q-D. Van²

¹ Siemens VDO, B.P. 1149, 31036 Toulouse, France. Arnaud.Albinet@siemens.com

² UTC/HEUDIASYC, UMR 6599, 60205 Compiègne, France. boulange,vanquang@hds.utc.fr

³ CEA, LIST, Boîte 94, Gif-sur-Yvette, F-91191, France. Hubert.Dubois@cea.fr

⁴ I3S, UNSA, CNRS/INRIA, B.P. 121, Sophia Antipolis, F-06903, France. map@unice.fr

⁵ INRIA, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France. Yves.Sorel@inria.fr

Abstract. We present a model-based methodology for requirements traceability proposed in the framework of the MeMVaTE_X project. The methodology relies on the EAST-ADL language and the two UML 2.0 profiles: MARTE for real-time embedded systems, and SysML for system requirements modeling. Along the paper, we illustrate the proposed methodology by an automotive case study, namely a knock controller, focusing on the time aspects of the requirements specified with the MARTE UML 2.0 profile. We explain how to define the requirements according to a proposed classification, and we present the tracing mechanisms based on the SysML UML 2.0 profile. Finally, we describe the proposed MeMVaTE_X methodology which extends the EAST-ADL methodology in order to take into consideration the expression of requirements, and their traceability along the life cycle.

Keywords: end-to-end traceability methodology, application of traceability, UML model driven process, automotive domain, real-time, embedded.

1 Introduction

Embedded applications found in automotive domain continually increase in complexity not only according to the functionalities they must provide, but also according to the requirements they must meet due to multiple constraints such as dependability, timing, resources, variability, etc. For these reasons it becomes necessary to trace these requirements all along the development life cycle leading to the intended product in order to guarantee they are met. One of the major difficulties is due to the modifications that requirements must undergo from the highest functional level to the lowest implementation level. Indeed, they must be as consistent as possible to guarantee that traceability is efficient.

The paper presents preliminary results of a work achieved within the framework of the MeMVaTE_X project⁶. This project is intended to provide a methodology for requirements traceability using a model driven engineering (MDE) approach in order to design

⁶ This work has been performed in the context of the MeMVaTE_X project (<http://www.memvatex.org>) of the System@tic Paris Region Cluster. This project is partially funded by the French Research Agency (ANR) in the "Réseau National des Technologies Logicielles" support.

embedded systems of the automobile domain. For this domain which demands a high level of dependability, sound methodologies are necessary to tackle the complex problems that arise. This project is closely related to other works carried out in the Competitiveness Cluster Ile-de-France System@tic, and in the European project ATESSST, since they also concern MDE for embedded applications as well as the automotive domain.

Nowadays, dependability is a critical issue in automotive systems. The automotive industry is now conceiving its own standard⁷ to introduce safety notions at every level of the development life cycle, from the system level down to the implementation level onto software and/or hardware. Since safety notions at each level are expressed by requirements, the control of safety-critical system depends on the control of requirements from their elicitation, their validation, and traceability through the different levels.

Requirement expression and traceability are key aspects of software engineering. The first studies on requirements engineering in the domain of software development has been started in 1990 [1]. Some tools like DOORS [2] handle the traceability management. More recently, the Paladin [3] approach proposed a requirement methodology based on UML in the domain of the web semantic. The SysML [4] UML profile allows now the designer to consider requirements as first class concepts in UML for system level design, and to deal with traceability concerns since relations between requirements and, requirements or model elements, are also defined in SysML. For the real-time embedded domain, the EAST-ADL [5, 6] language proposes a way to integrate requirements in the modeling approach process for automotive systems. Nonetheless, EAST-ADL neither covers all the requirement classes nor their traceability, and does not propose an integrated methodology.

Therefore, we propose a MDE methodology based on UML and its extensions (MARTE [7], EAST-ADL, SysML) for the modeling of requirements and their traceability in embedded systems. In this paper, we shall only focus on the three first levels defined in the methodology associated with the EAST-ADL language. However, it is planned that the five levels of the development life cycle will be covered at the end of the MemVaTeX project. Consequently, the EAST-ADL methodology is extended in order to take into consideration the expression of requirements, possibly of different types, and their traceability along the development life cycle. An important issue concerns the transformation of the requirements when one proceeds to a next level because they request that corresponding rules are defined. We model the requirements and their traceability with the SysML UML profile that was defined for this purpose, while of course, relying on the EAST-ADL levels. Since timing constraint issues are of crucial importance for the dependability of the applications we are interested in, we use the MARTE UML profile to model accurately time relationships through the different levels of the development cycle. For other issues such as dependability, safety, availability, or security we merge the interesting features of SysML, EAST-ADL and MARTE.

We illustrate the proposed methodology by using an automotive case study, namely a knock controller.

The paper is structured as follows. Section 2 is an overview of the EAST-ADL language, and the EAST features we adopt in our methodology. Section 3 presents the knock control system focusing on the temporal characteristics. Section 4 focuses on the

⁷ The IEC 26262, derived from IEC 61508.

requirement expression classification and modeling. The methodology is presented in Section 5. Some perspectives of this work are given in the conclusion.

2 The EAST-ADL process

EAST (Embedded electronic Architecture Study)-ADL (Architecture Description Language) is a language for the development of vehicle embedded electronic systems. EAST-ADL was developed in the context of the EAST-EEA European project. It provides a unified notation and a common development process for all the actors of a car development (car-maker, suppliers, . . .). EAST-ADL allows a decomposition and a modeling of an electronics system through five abstraction levels (*Vehicle*, *Analysis*, *Design*, *Implementation*, and *Operational*). These levels and the corresponding model elements provide a separation of concerns.

The *Vehicle* level describes the main functionalities, and the variability points of the vehicle (Stakeholders view). The *Analysis* level models and refines these functionalities, and their interactions (Control/Command engineers view). The *Design* level represents a decomposition of functionalities with respect to allocation constraints, reuse, mode change, etc. (software engineers view). The *Implementation* level is an instantiation of the design level model. It produces a flat software structure which takes into account the software parts, the protocols and the OS (design engineers view). The *Operational* level consists in mapping the implementation model onto the effective ECU, frames, and tasks (design engineers view). An example of complete prototype car developed with EAST-ADL process can be found in [8].

Another objective of EAST-ADL is to propose mechanisms to support variant handling, requirement expression, and requirements traceability. The requirements in EAST-ADL are an extension of those of SysML. They can be modeled either as a textual description or using a formal description, and they can be attached to create a dependence with any EAST-ADL objects. Requirements traceability is also based on the same dependencies of SysML.

Today, EAST-ADL does not provide an integrated framework offering all these interesting aspects. Our methodology inherits from the EAST-ADL process, the different abstract levels (*Vehicle*, *Analysis*, *Design*, *Implementation*, *Operational*), but it enriches the EAST-ADL language with some model elements such as the expression of time, and the resources allocation. The traceability and requirements aspects follow a SysML syntax.

3 Case study: knock controller

We illustrate our methodology with the support of an automotive application: the detection and control of the knock phenomena. In gasoline internal combustion engines with spark ignition, an undesired effect may occur when the fuel mixture partially automatically ignites as a result of the compression in the combustion chamber.

Figure 1(a) shows the origin of the knock phenomenon. In a gasoline internal combustion engine, when the piston compresses the mixture, the spark plug produces a

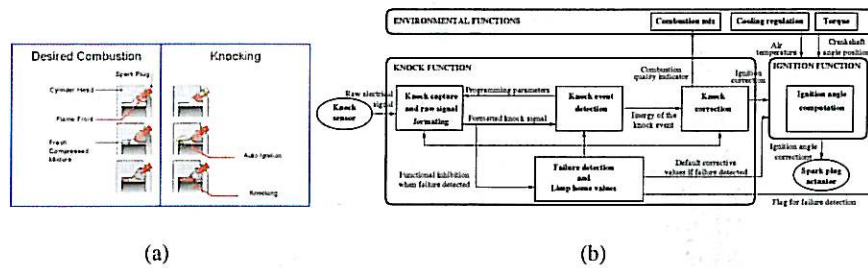


Fig. 1. Knock controller description.

flame front in the combustion chamber. The generated electric spark ignites the mixture, and produces uniform waves that push the piston to its original position. The very rapid heat release implied by this abnormal combustion generates shock waves that: decrease the engine performance, increase some pollutants that do not comply with norms, generate undesired engine vibrations, and decrease the user comfort. Eventually, it may cause a destruction of the engine, by damaging the spark or piston, potentially leading to jamming.

For these reasons, monitoring the knock phenomenon is a critical issue. An appropriate anti-knock control, represented in Figure 1(b), is applied to each cylinder at every engine cycle from low engine speed up to the highest engine speed. A knock control system consists of one or several noise sensors, and a controller which acquires the noise through the sensors, and computes the correction during the combustion phases of the cylinders. Due to the speed of the combustion cycle in a motor, these treatments may be handled while satisfying complex real-time constraints during the design of the corresponding function. The knock control function equips a very wide range of engine management system for gasoline system. Several strategies for acquisition and correction are possible for optimizing the treatment of the knock, leading to manage a lot of variants for this function.

All these variants must be handled at the early stage of the design process. They lead to multiple solutions at the end of the design process. A first-part solution to manage this complexity consists in proposing a precise expression and classification of the requirements, using adapted models.

4 Expression and classification of requirements

Requirements are generally expressed by stakeholders or automatic control and software engineers with natural languages. In a design process, requirements must be validated and verified. On the one hand verification means that the designer must guarantee, at the different abstraction levels, that the requirement has been taken into account, and that they give rise to a corresponding model element. Verification is of particular importance in a certification process. On the other hand validation means that the designer

must guarantee, at the different abstraction levels, that the model satisfies the requirements.

We focus here on the verification of requirements. Since we develop a methodology based on the EAST-ADL process, requirements must be expressed at the different abstraction levels of this process. They must be refined, and a link must exist between requirements expressed at the different levels. A link between requirements is expressed using the traceability mechanisms of SysML. We propose a classification and a labeling of the requirements. These features permit the characterization of traceable paths for a requirement, or a class of requirements, through the levels of the EAST-ADL process. In the next section we present the different classes of requirements we have adopted. With the support of the knock example, we illustrate the SysML mechanisms provided for traceability, and show how to associate requirements to model elements of the methodology. For this purpose, we use the MARTE UML profile for addressing the expression of real-time requirement.

4.1 Definition and classification of requirement

The design of embedded real-time systems requires a precise management and tracing of user's requirements. It becomes even more critical when the design process must comply with a standard such as the IEC 880 in nuclear system, CENELEC EN 50126, EN 50128 and EN 50129 in railway system in Europe, and DO-178 and DO-254 in aeronautic system. Above all these standards there is the general IEC 61508 standard [9] which concerns all systems based on electric, electronics, and programmable electronics. These standards recommend the application of requirement engineering with end-to-end traceability applied to the whole V-cycle. Details and discussions about these standards can be found in [10].

Requirements are generally expressed in natural language. Some research initiatives are provided to transform the expression of needs into a set of requirements. A simple description of a requirement is not fully sufficient to define it. There is other status information that each requirement must carry. The requirements must be tagged to provide such information.

For that purpose, we consider a requirement as a structure with several attributes. A requirement is characterized by an *identification*, a *textual* description, and a *type* (*functional*, *non-functional*). The non-functional type is refined into sub-types such as reliability, performance, safety, and cost, etc. Some other attributes indicate the *derivation type* (decomposition/refinement), the document source where the origin of the requirement can be found, the *verification method* that must be applied on this requirement, and its *agreement status*. Some of these attributes are automatically generated. For example, the identification attribute consists of a number, and a label representing the level in the EAST-ADL process (Vehicle Level VL, Analysis Level AL, Design Level DL, etc.) Other attributes are defined by the user (Functional/Non-Functional), others attributes are flags which are set after an analysis phase (Agreement status).

Table 1 gives some examples for the identification and definition of requirements for the knock controller.

This precise labeling of requirements becomes essential when a traceability process is requested. It is important to demonstrate that all input requirements are satisfied at

Table 1. Example of requirement expression.

Req. Name	ID	Text
Eng.Pha.Pos.	VL-F-2	The engine management must detect the different positions of the cylinder and control the ignition in an optimal manner.
Eng.Cam.Pos.	AL-F-4	Observing the camshaft position, the knock function must locate the ignition phase in a 4 stroke cycle.
Eng.Cra.Pos.	AL-F-8	Observing the crankshaft position the knock function must detect which cylinder is concerned by the knock correction.
Kno.Con.Dur.	AL-NF-P-2	The filtering and the detection/correction must be executed before the next ignition phase of the same cylinder.
Acq.Dur.Con.	AL-NF-P-3	The acquisition duration window must be large enough to acquire at most 2 samples of the knock sensor.
Rot.Spe.Val.	AL-NF-D-2	The rotation speed for the motor is lower than a constant MAXRPM.

the lower levels (i.e. they are linked to other requirements, or they have been taken into account by a model). This demonstration is based on links established between requirements, and on arguments associated to these links.

We use SysML to express the requirements and their relationships. The section 4.2 presents the different constructions taken from SysML for this purpose.

4.2 SysML requirement and tracing mechanisms

The SysML profile (for Systems Modeling Language) is a specialization of UML for systems engineering applications. SysML supports specification, analysis, design, verification, and validation of various systems potentially complex. SysML is defined as an UML profile since it uses a subset of UML 2.1 [11]. SysML extends UML with new notations and diagrams such as the requirement, and parametric diagrams which are an extension of internal block diagrams. We focus on requirement diagram since this is the most appropriated for our considerations.

The purpose of requirements in SysML is to provide a relationship between requirements, as traditionally managed, and model elements as usually managed in UML. Thus, a requirement is a stereotype of a UML class that is subject to a set of constraints. A requirement in SysML is composed of an identifier and a text that describes the requirement in a natural language. Additional properties can be attached to these two fields. We use this functionality in our approach.

Several links are defined in SysML to express requirements relationships, and to link them to other model elements. The different relationships are the requirements hierarchy (*refine*) to describe how a model element can be used to further refine a requirement, the derivation (*derive*) that corresponds to requirements at the next level of a hierarchy, the requirement satisfaction (*satisfy*) which describes how a design or an implementation model satisfies a requirement, and the verification (*verify*) which defines how a test case verifies a requirement. With such relationships, initial requirements can be traced by following a top down path in the requirement tree. In reverse, a requirement at any level can be traced back to its origin. Supported by these previous SysML relations,

traceability between requirements and model elements, are considered directly in the models; such as traceability between requirement evolutions in the model development. The Figure 2 illustrates a requirement diagram and relationship between requirements and model elements for the knock controller case study.

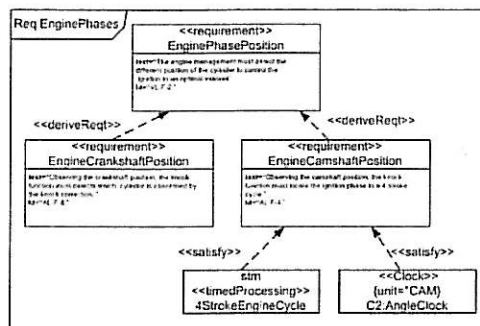


Fig. 2. Example of a SysML requirement diagram.

4.3 MARTE profile for real-time requirements modeling

Real-time constraints belong to the non-functional requirements class. The MARTE profile is used in conjunction with SysML to model the temporal and allocation characteristics of an embedded system. MARTE extends the modeling capabilities of UML and SysML, and makes clear the semantic with an objective of model validation.

In this section we pay a special attention to the time model of MARTE which allows the modeling of multi-clock systems. As embedded systems interact with mechanical and physical components the software computing part is usually triggered by heterogeneous events. A clock can be either associated with the classical time (second, hertz) or a logical one (round per minute, angular position, meter).

The knock control example is a good candidate for multi-clock system modeling. The capture phase depends on the period, in hertz, of the acquisition sensor whereas the filtering and correction phases are triggered by events the occurrences of which are measured in angle degrees. This possibility to deal with such logical time is adapted to the specification expression of embedded systems. It allows a manipulation of time at a high level of abstraction, and a direct relationship between the requirements and the model. The definition of clocks in MARTE corresponds to the definition of a class named AngleClock which is the time base that represents the temporal evolution of a rotation. From this class we declare two instances of this clock, camClk and crlClk. These clocks represent the revolution position of the camshaft and the crankshaft which are crucial mechanical elements of an engine. The definition of these two clocks participates to the satisfaction of the AL-F-4 and AL-F-8 requirements (cf. Figure 2). These two clocks have a resolution, an offset, and a maximal value (modulo).

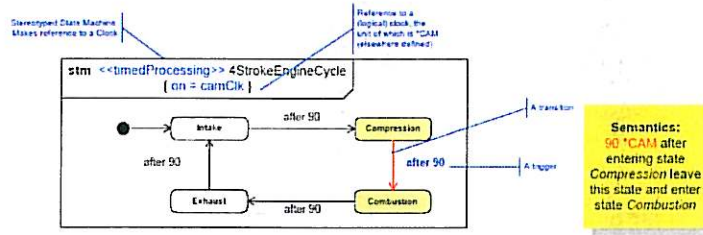


Fig. 3. State machine of a four stroke cycle.

These clocks can trigger some processing as it is showed in Figure 3. The timed-Processing elements are UML behaviors that can be triggered by logical events or clocks. Some duration constraints may be expressed on these behaviors. For example, the requirement AL-NF-P-2, which belongs to the performance class, is satisfied by a duration that must be associated as a constraint to the knock controller block. $\ll\text{timedDurationConstraint}\gg\{K\text{Filter.Duration}+K\text{Detect.Duration}<720^\circ\text{CAM}\}$

This capability to express at the same level of abstraction the requirements and the model is a way to bridging the gap between the requirements and the model, but also to link the requirements with the underlying properties to be verified.

5 MeMVaTEx methodology

We propose a methodology that considers requirements as a first class concept, from the early steps of modeling, and during all the phases of the development life cycle. The MeMVaTEx methodology aims at merging different languages and standards: UML2.0 and SysML are used for system modeling and requirements traceability in the models, the EAST-ADL language is used for the automotive architectural description. The MARTE UML profile deals with the real-time constraints modeling. Furthermore, the EAST-ADL process is adopted as a standard to provide the different modeling abstraction levels (see section 2).

As shows in Figure 4, the proposed methodology keeps the five levels of the EAST-ADL process for structuring the development. At this stage of the project we have only addressed the vehicle, analysis, and design levels. For each level, two branches evolve in parallel. The requirement branch on the left side allows expressing, defining, and tracing the requirements of the system. Requirements are expressed in such a way that they can be managed by dedicated tools; the modeling branch on the right side is composed of models integrating related requirements. A set of heterogeneous models can be explored at each level for expressing different parts of the system (behavior, architecture, algorithms, allocations,...) For instance, UML/SysML models are built with the ARTiSAN Studio tool, and the behavioral functions with the Simulink tool. In order to manage the requirements we use Reqtify, a light and powerful tool who facilitates the traceability: textual requirements – stored in Word or Excel documents – are imported to Reqtify [12], and are represented in our model. Each requirement is linked to another

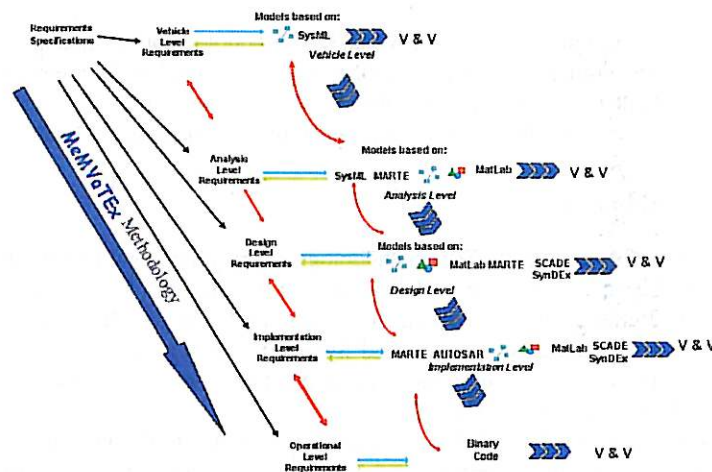


Fig. 4. The MeMVaTeX methodology.

requirement, or a part of the model, or a test case. Reqtify manages these links and can export analysis and traceability reports at any level during the development process.

The general outline presented in Figure 4 shows that initial requirements are given through a textual form, and must be dispatched at the different levels. The originality lies in the fact that these requirements are considered along the whole application development life cycle, and built at each level. Additionally, some requirements can appear at intermediate levels, like exported requirements, and for these reasons, it can be confusing to trace back these requirements to former levels. Requirements are traced among the modeling branch by using the traceability mechanisms offered by the SysML metamodel (composition, verification, satisfiability, etc.) This traceability can also be followed between the corresponding requirements in the requirement documents.

From this point, requirements models are defined, and traceability is achieved between requirements pools and models. This is done by building references between requirements elements included in the models and requirements in external documents. These external documents (on the left branch in the Figure 4) are requirements pools. In the project, the references between the initial requirements documents and the requirements model elements are managed by traceability tools like Reqtify or its open-source release in Topcased, TRAMWAY.

Model elements used at each level are carefully selected in the methodology. We have voluntarily limited the use of some models because they are not endowed with a sufficiently precise semantics giving rise to ambiguous models. All these choices are made with the objective to connect the models with validation and verification (V&V) tools. These tools will check that models satisfy properties induced by requirement expressions. The connection with V&V tools and technologies will be facilitated by the MARTE profile that includes the analysis part of UML models for real-time systems; it will also be facilitated by the use of Matlab models and its connection with the Simulink

tool for the design levels. The MeMVaTeX methodology achieves a major goal for industry in the domain of system and software engineering based on model driven architecture. It allows the designer to facilitate the use of common tools, an easier update and evolution of the models, and the requirement integration along the MeMVaTeX process.

6 Conclusion

We presented the MeMVaTeX methodology for requirements traceability through the first three levels of the EAST-ADL process in the case of a knock controller, example coming from the automotive domain. Relying on the two UML 2.0 profiles: MARTE for real-time embedded systems, and SysML for system requirements, it allows the designer to express the requirements according to a proposed classification, and to trace them along the EAST-ADL process.

This method will provide a competitive advantage to the industry, a mastering of the system development quality, and will reduce the cost of re-engineering by decomposition of solution, and backward impact analysis (reuse) centered on the requirement management.

As future works we plan to extend the methodology to the five levels of the EAST-ADL process taking into account the requirement related to the hardware architecture, the operating system, and the allocation of functions to the hardware components.

References

1. Thayer, R.H., Dofman, M.: System and Software Requirements Engineering. IEEE Computer Society Press Tutorial, 1990.
2. Hull, E., Jackson, K., Dick, J.: Requirements Engineering. Springer, Second Edition, 1995.
3. Mayank, V., Kositsyna, N., Austin M.A.: Requirements Engineering and the Semantic Web. Part II. Representation, Management and Validation of Requirements and System-Level Architectures. ISR Technical Report 2004-14, Univ. of Maryland, College Park, 2004.
4. Object Management Group. The Systems Modeling Language, 2006. <http://www.sysml.org>
5. ITEA Project Version. EAST-ADL: The EAST-EEA Architecture Description Language. 2004. <http://www.east-eea.net>
6. Debruyne, V., Simonot, F. and Trinquet, Y.: EAST-ADL an architecture description language – validation and verification aspects. IFIP, WADL04, 2004.
7. Object Management Group. UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE). Request For Proposals, 2006.
8. Lönn, H., Saxena, T., Nolin, M., Törngren, M.: FAR EAST: Modeling an Automotive Software Architecture Using the EAST ADL. ICSE Workshop on Software Engineering for Automotive Systems, Edingbourg, 2004.
9. International Electrotechnical Commission. Functional safety of electrical/electronic/programmable electronic safety-related systems. Std. 61508, 2000.
10. Boulanger, J-L.: Expression and Validation of Logical and Physical Safety Properties for Critical Systems, PhD thesis (in French), Université de Technologie de Compiègne, 2006.
11. Object Management Group. Unified Modeling Language: Infrastructure, version 2.0, Document formal, 2006.
12. Laronde, E., Burgaud, L., Fourgeau E.: Shift towards a Cohesive Design Based Management of Automotive Embedded Systems Requirements. European Congress on Embedded Real Time Software, 2006.

Traceability as Input for Model Transformations ^{*}

Bert Vanhooff, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers

Department of Computer Science, K.U.Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium
{bert.vanhooff, stefan.vanbaelen, wouter.joosen,
yolande.berbers}@cs.kuleuven.be

Abstract. Some model transformations require more information than can be derived from its source model(s) in order to generate a meaningful target model. For example, a transformation with two source models needs to know how their respective model elements relate; these relations often only exist implicitly as part of the transformations developer's knowledge. In this paper we show that traceability models, who can be automatically generated as part of any model transformation, contain explicit inter- and intra-model relations that are valuable to subsequent transformations. We explain how to extract this information and propose a number of additions to current transformation techniques that are needed to completely open up traceability information to transformation developers.

1 Introduction

Classically, the creation and maintenance of traceability information has been a manual and labor intensive task. The rise of Model Driven Development (MDD) eases this problem by introducing model transformations, which can generate basic traces automatically. MDD recognizes the need of many different intermediate models to represent a system, which results in an abundance of readily available traces that interrelate every piece of the system. Typical uses of this traceability information are: showing that requirements are met, analyzing impact of changes, propagating changes, etc.

In this paper we present a new use of traceability information in the context of transformation chains. A transformation chain or transformation composition is a network of many subtransformations, each contributing a small part to a larger transformation goal. Each subtransformation also produces traces that contribute to a *global traceability graph*. This graph interrelates disparate models that represent different aspects of a system (e.g. functional, security and persistence model), produced as part of a transformation chain. In some cases we need to know exactly how these aspects are related in order to perform further transformations (e.g. which database table stores a particular attribute from the functional model). This kind of information usually cannot be extracted from the individual models but is hidden in the traceability models.

We will show how generated traces can provide non-trivial inter- and intra-model relations without imposing many additional requirements on the models or transformations. We define *trace navigation* as an operation to derive the required information

^{*} The described work is part of the EUREKA-ITEA MARTES project, and partly funded by the Flemish government institution IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders).

and propose *trace tagging* to enrich the information contained in the traceability graph. Finally we discuss how these concepts can be integrated in current transformation technologies.

The rest of the paper is structured as follows. Section 2 provides the necessary background on traceability modeling and trace generation. In Section 3, we show that generic traceability information can be used to derive non-trivial relations between model elements. In some cases, we must add a specific meaning to traces in order to extract useful information (Section 4). In Section 5 we discuss how our approach can be integrated in current transformation technologies. We wrap up by presenting related work (Section 6) and drawing conclusions (Section 7).

2 Background

In the context of MDD, traceability information is represented as a model on its own. In this section we briefly discuss what a traceability (meta)model is (2.1) and how traces can be generated automatically by transformations (2.2).

2.1 Traceability Metamodel

A traceability metamodel captures how model elements may be related by trace relations and defines the semantics of such relations. It is often necessary to distinguish between different kinds of traces. For example, it is interesting to know whether a trace denotes a relationship between a textual requirement and an architectural element or indicates a refinement within the design. The required kinds of traces are often highly project dependent [1]. In this subsection, we make a concise comparison between two different traceability metamodels that both facilitate specific kinds of traces.

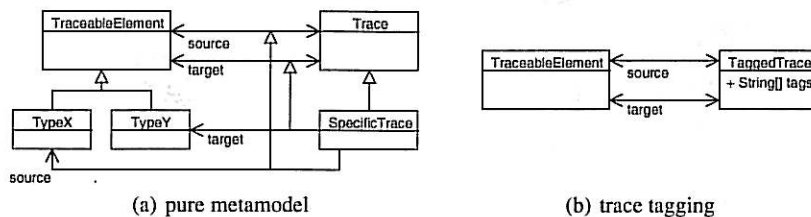


Fig. 1. Two extreme traceability metamodels.

In a *pure metamodel* approach, the traceability metamodel specifies every kind of trace we may need. Figure 1(a) shows a snapshot of such a metamodel. The two top classes indicate that a *Trace* can relate two *TraceableElements*. If we need a more specific kind of trace it suffices to subclass *Trace* and specialize its associations (see *SpecificTrace* in the figure). The most notable advantage of this approach is that we can precisely specify both usage constraints and semantics of each trace kind at the metamodel

level. The main drawback is its lack of flexibility: every change in traceability requirements needs to be reflected (hard-coded) in the metamodel. Project-specific trace kinds are expressed directly in the metamodel, which limits potential reuse to one project.

The *trace tagging* approach uses a simple and general traceability metamodel (see Figure 1(b)) and allows users to annotate generic traces with attributes or *tags*. A similar approach is used in specialized tools such as Telelogic DOORS. The kinds of traces are represented by tags and are defined at the model rather than at the metamodel level. This prevents us from specifying precise semantics and usage rules in the metamodel. A user can add any tag without having to adhere to any rules. At the same time this kind of flexibility is its biggest advantage. The metamodel never needs to be changed, hence this kind of metamodel can be reused in any project.

We will show that, because of its generality and flexibility, the trace tagging approach is best suited for integrating traceability into transformation compositions.

2.2 Automatic Trace Generation

Automatic model transformations can generate traceability information along with the target model(s). According to [2], transformation approaches either have dedicated support for traceability or rely on the developer to encode traceability as a regular output model. In any case, it is favorable to incorporate traceability generation into the transformation as opposed to producing it manually, no matter whether we apply the pure metamodel or tagging approach.

The major advantage of dedicated traceability support is that we get the traceability model(s) at an extremely low cost; the developer has to do little to no additional effort. A practical disadvantage is that the traceability metamodel is then fixed and may not be standardized among different transformation engines. Also, the level of granularity of the traces may not be the same.

Alternatively we can treat traceability as a regular output model of the transformation and incorporate additional transformation rules to generate it. The choice of metamodel is then completely at the discretion of the developer and does not depend on the transformation engine. The drawback is that additional effort is required to add traceability-specific transformation rules, which also pollute the implementation. An approach that partly solves these issues by automatically generating the trace-related transformation rules was proposed in [3].

Since our research is focussed on composing many subtransformations and combining different transformation technologies, we can only assume a generic traceability metamodel (i.e. Figure 1(b)) as the least common denominator of all transformation technologies. Furthermore we assume that each transformation creates a separate trace for each generated target model element; this link points to the source element(s) from which the target was created. Hence, no model element exists that is not fully traceable back to its source throughout the transformation chain.

3 Leveraging Traceability Models

In an MDD approach we use many models that offer different views on a system: different levels of abstraction, different sub-aspects of a system, etc. Many of these models

are derived through automatic transformations and are hence highly interlinked through automatically generated traceability models (as discussed in 2.2). In 3.1 we show that the information contained in traceability models can be leveraged for development of further transformations. We discuss an elaborate example in subsection 3.2.

3.1 Discovering Useful Trace Information

Figure 2(a) and 2(b) present examples of transformations and models along with their generated traceability models. Each curved line in the figure represents a single trace and is part of a traceability model. For example in Figure 2(a) there are in fact three models: UML-1, UML-2 and the traceability model that connects these two.

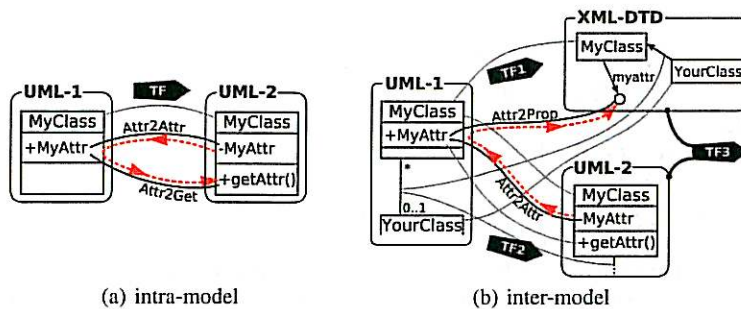


Fig. 2. Deriving model relations from past traceability information.

Figure 2(a) shows a transformation TF from source UML-1 to target UML-2 that adds accessor methods (i.e. getters) to classes and makes corresponding attributes private. We focus on traces *Attr2Attr* and *Attr2Get*, labeled according to their transformation rule. The dotted arrows on the figure indicate that we can navigate back and forth through the traceability model and the UML-1 model to discover an *intra-model* relation between *MyAttr* and *getAttr()* in the UML-2 model. A subsequent transformation can use this information, which matches attributes to their accessors, for example to generate a detailed implementation for accessors. As a result, the concern of naming the accessor and making the attribute private (transformation TF) can be separated from the implementation generation. The necessary information is implicitly passed via the generated traces. Without traceability, a subsequent transformation would only be able to guess the relation between accessor and attribute (e.g. by matching strings).

A second example is shown in Figure 2(b). Class model UML-1 is transformed into an XML data type definition model (XML-DTD) [4] and a refined class model UML-2. This transformation is accomplished in two parallel steps (TF1 and TF2); TF2 is equivalent to the transformation in Figure 2(a). Suppose that we now introduce an additional subsequent transformation (TF3) to generate a persistence layer that can load objects from an XML file. This transformation needs to know, amongst others, the exact relations between class attributes and XML properties in order to produce a valid target

model. If we solely use UML-2 and XML-DTD as inputs it is hard to find these relations. Resorting to the traceability models offers a solution. For example, the XML property that corresponds to class attribute *MyAttr* can be found by navigating back (*Attr2Attr*) and forth (*Attr2Prop*) through the traceability model (see dotted arrows). This example shows that we can also derive *inter-model* relations from traceability models.

In the former paragraphs we have used ‘navigate’ whenever we combined the information from one or more traces to derive relations between transformation input model elements. We consider the union of all traceability models and regular models as a graph G where elements from the regular models are vertices V and traces are edges E . We can hence give a definition for trace navigation.

Definition 1. *Given an input vertex v_i and a set of navigation target models M_t , a trace navigation is any operation that takes v_i as input and yields a collection of vertices V_o as output. All the elements of V_o are reachable from v_i and are owned by a model in M_t .*

3.2 Example: Persistence with a Relational Database

In this subsection we present a realistic example where a relational persistence layer is generated by a composition of a transformations. The complete composition is shown in Figure 3.

In the upper part, TF_{A1} and TF_{A2} subsequently transform the initial class model into an entity-relationship model (ER) [5] and corresponding relational database tables (RDB/SQL). Notice that the transformation from entity-relationship to database tables (TF_{A2}) is not a trivial one-to-one mapping; only two tables are introduced to represent three elements of the ER model. In the lower part, TF_{B1} and TF_{B2} transform the class model into a simplified class model UML-2 (taking away the association class) and subsequently a JAVA model. The figure also shows all the traces that were created by each subtransformation.

Having this many intermediate models has the advantage that different concerns are treated separately in different transformations and are visible in separate, highly specialized models. This narrows the scope of each subtransformation, making them easier to implement, and offers a dedicated view to each domain expert – i.e. a database specialist considers the ER model while a Java programmer only looks at the JAVA model.

After executing all the transformations, we end up with the *current* models: RDB/SQL and JAVA. At this point we introduce a new transformation TF_{NEW} (see Figure 3) that adds database access code to the JAVA model; part of the resulting model is shown in the rightmost part of the figure. Transformation TF_{NEW} takes both RDB/SQL and JAVA models as input since a persistence layer involves getting values from the tables in the database (RDB/SQL model) and storing them as class attribute values of the JAVA model.

In Figure 3 we show how we can relate classes to database tables in order to generate the correct sql query for the *Person* class. Just by navigating the traces through the old models, we end up with the *Employee* table (follow the lines marked with dots), which is indeed the table where the information about *Person* is stored; note that we also get the corresponding primary key. It would be very difficult to find this relation without the traceability information since there is no one-to-one correspondence from the RDB/SQL

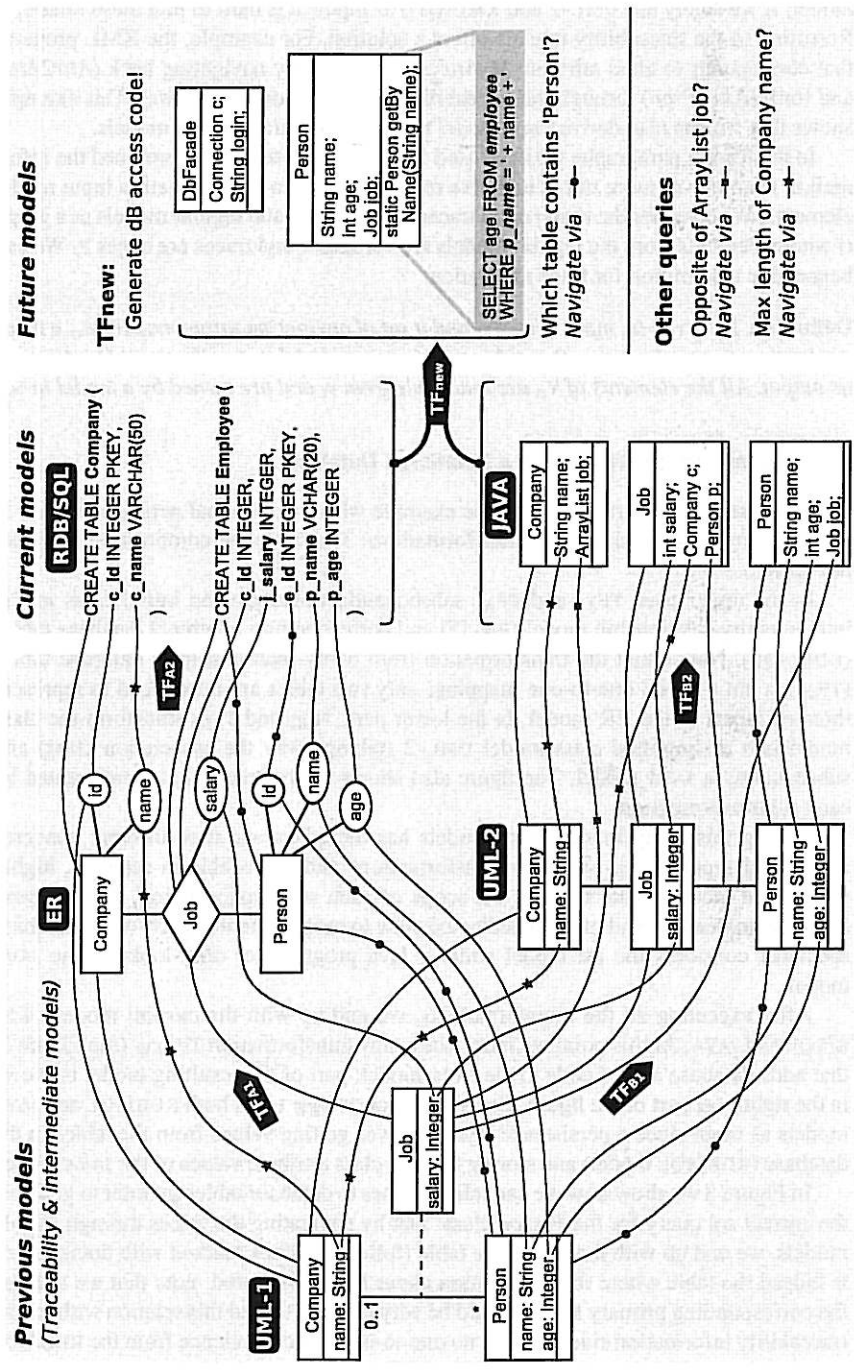


Fig. 3. Using traceability in a transformation chain to generate a relational persistence layer.

model structure to the JAVA class structure. It is easy to see that we can also use the traces to find corresponding table fields for the attributes of the *Company* and *Job* class, for example a *Job*'s *salary* attribute corresponds to *j_salary* column in the database. Because of space restrictions we leave it up to the reader to find other relations hidden in the traceability information. Two other examples are given at the bottom rightmost corner of Figure 3.

In this and the previous subsection we have shown how we can extract valuable information from a global traceability graph that cannot be extracted from models individually but is sometimes required to perform meaningful transformations. A key observation that can be made at this point is that, although we heavily use the traceability models to derive relations, a transformation only needs to consider its actual inputs. The navigation through the traceability models can be handled transparently by a generic 'trace query' operation. As a consequence, a transformation does not have to depend explicitly on each one of the previous (traceability) models (see also Section 5).

4 Producing Unambiguous Traceability Models

As allowed by Definition 1, a trace navigation can return more than one result. In many cases the required element can be selected by filtering on the element's type (see also Section 3.2). If the result of a navigation yields several elements of the same type however, the obtained information will be ambiguous. We retake the example where an accessor method is generated for each class attribute and we now include generation of mutator methods (i.e. a setters) in a separate transformation (see Figure 4). If we start navigation from attribute *MyAttr*, we get a collection of two operations as result: both mutator and accessor (see dotted arrows). Since they are of the same type (*Operation*), it is impossible to filter out the desired result.

To solve this issue we need to add additional semantics to the traces that allow to distinguish different trace paths. In case of the example we need to be able to find out which trace path leads to the accessor and which leads to the mutator. Since we use a generic traceability metamodel, an appropriate way to add these semantics is by applying the trace tagging approach (explained in 2.1). This way we do not limit the possible semantics of the traces by the traceability metamodel; these semantics are very dependent on the type of transformation and it seems quite impossible to capture all possibilities a priori as in a pure metamodel approach (see 2.1).



Fig. 4. Navigating tagged trace links.

In figure 4 we apply two consecutive transformations: first we add mutators (TARGET1) and subsequently we add accessors (TARGET2) and tag the traces appropriately

with 'set' and 'get'. Further transformations can now unambiguously find both mutators and accessors for each attribute in model TARGET2 by navigating via the appropriate traces. As long as a 'set' trace is encountered on the navigation path we get the mutator; when a 'get' tag is encountered, we get the accessor. We refer to a navigation that takes tagged traces into account as a qualified navigation. The following definition uses the same assumptions as Definition 1.

Definition 2. A *qualified trace navigation* for tag t is a navigation where each path from input vertex v_i to any of the result vertices in V_o contain at least one edge (trace) tagged with t .

In case the tagged trace models, produced by previous transformations, still yield ambiguous trace navigation results it is up to the transformation assembler (who specifies a transformation composition) to decorate the traces with additional tags.

5 Integrating Traceability in Transformations

We have shown that automatic model transformations can generate traceability models at a very low cost and that we can navigate through traces to extract useful information. In this section we identify three transformation areas that need to be extended in order to fully implement our approach. We also propose early ideas to extend these areas.

Production of Traces It is not very hard to produce traceability models as part of model transformations. Some transformation languages even offer dedicated support for this purpose (for an overview, see [2]). Nevertheless, we believe that it would be useful to see how this can be improved, keeping the trace tagging metamodel in mind. Adding dedicated trace tagging syntax to transformation languages could be very useful, for example allowing annotation of each transformation target element with a tag that is transferred to a corresponding trace at execution time (see Listing 1.1).

```
rule Attr2Set
  source( Attribute aIn )
  target( Operation oOut ) tag 'set'
```

Listing 1.1. Example syntax for trace tagging (in pseudo code).

Traceability Queries In order for transformations to access traceability information (through the global traceability graph), they will need to include both traceability models (edges) and past regular models (vertices) as additional inputs. This has very negative effects on various aspects of the transformation. First of all reusability goes down since the transformation is directly dependent on all the intermediate results produced by previous transformations. Secondly, the specification of the transformation becomes cluttered with many additional models that are not used directly in the transformation but are solely used to derive relations between the real input models. Finally, trace navigation needs to be encoded in the transformation itself, cluttering its implementation.

However, as briefly mentioned in Section 3.2, a transformation does not have to depend directly on all the previous models if we factor out the (qualified) trace navigation operation. We propose to introduce the *traceVia(tag: Set(String)): Set(oclAny)* operation in OCL as the single access point to the traceability information. In Listing 1.2 we use this operation to find out if an operation is a mutator. This approach would eliminate most of the disadvantages described in the previous paragraph.

```

rule RenameMutators
  source( Operation in )
    when in.traceVia( 'set' )->select(oclIsTypeOf( Property ))->
      notEmpty()
  target( Operation out )
    out.name = 'my' + in.name

```

Listing 1.2. Example syntax for trace navigation (in pseudo code).

Declaration of Traceability Usage Each transformation needs to have a clear specification so that it can easily be reused and composed in a transformation chain. In the first place, this comes down to a specification of in and output models. But, if transformations also depend on traceability information from previous transformations, the specification also has to include that information. We think that this can be accomplished by specifying required and provided trace tags. Each transformation can define its own tags independently of other transformations, improving reusability. Whenever the transformation assembler connects transformations it might then be necessary to do a semantic mapping between required and provided tags. More research is needed to find a suitable set of mapping strategies.

6 Related Work

In previous work [6] we have proposed a technique that uses UML Profiles to encode intra-model traceability relations directly in a UML model. The approach taken in this paper extends this work by externalizing traces, allowing for inter-model relations and metamodel independence.

Most current transformation languages [7, 3, 8] build an internal traceability model that can be interrogated at execution time, for example, to check if a target element was already created for a given source element. This approach is specific to each transformation language and sometimes to the individual transformation specification. The language determines the traceability metamodel and the transformation specification determines the label of the traces (in case of QVT/Relational the traceability metamodel is deduced from the transformation specification). The approach taken only provides access to the traces produced within the scope of the current transformation. The aim of our approach is to open up this information to other subtransformations.

Marvie describes a transformation composition framework [9] that allows manual creation of *linkings* (traces). These linkings can then be accessed by subsequent transformation, although this is limited to searching specific traces by name, introducing tight coupling between subtransformations. Our proposal allows a more loose coupling

by matching tags within a trace navigation and is able to deduce information that spans more than a single trace.

In [10], a traceability framework for Kermeta is discussed. This framework supports the creation of traces throughout a transformation chain. However, the authors do not discuss how this information can be exploited in subsequent transformations.

7 Conclusions & Future Work

Currently, most model transformations do not take the results of previous transformations into account. These intermediate results are a collection of regular and traceability models; the latter can be produced at very low cost in an MDD setting.

In this paper, we considered intermediate models as part of a *global traceability graph* and showed that this graph can be used to derive useful relations between (inter) an within (intra) input models of subsequent transformations. These relations cannot be derived from individual input models but are sometimes required to specify meaningful transformations. We defined the notion of (*qualified*) *trace navigation* as a meta-model independent way to search for such relations. We introduced semantic annotations (*tags*) on the traceability models to avoid ambiguities during trace navigation.

Finally, we discussed three areas in transformation techniques that need to be extended in order to enable the use of traceability in transformation development. These are integration of trace creation (1) and trace navigation/querying (2) in transformation languages and (3) declaring traceability usage in a transformation specification. We are currently extending ATL [3] with the *traceVia* operation that we proposed as a solution for (2). We will use this extended version of ATL to further validate our approach.

References

1. Gills, M.: Survey of traceability models in it projects. In: ECMDA-TW Workshop. (2005)
2. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA 2003 WS on Generative Techniques in the context of Model Driven Architecture. (2003)
3. Jouault, F., Kurtev, I.: Transforming models with atl. In: Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica (2005)
4. Ceri, S., Comai, S., Damiani, E., Fraternali, P., Paraboschi, S., Tanca, L.: XML-GL: A graphical language for querying and restructuring XML documents. In: Sistemi Evoluti per Basi di Dati. (1999) 151–165
5. Thalheim, B., Thalheim, B.: Entity-Relationship Modeling: Foundations of Database Technology. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2000)
6. Vanhooff, B., Berbers, Y.: Supporting modular transformation units with precise transformation traceability metadata. In: ECMDA-TW Workshop, SINTEF, (2005) 15–27
7. Object Management Group: Qvt-merge group submission for mof 2.0 query/view/transformation. Misc (2005)
8. Lawley, M., Steel, J.: Practical declarative model transformation with tefkat. In: MoDELS Satellite Events. (2005) 139–150
9. Marvie, R.: A transformation composition framework for model driven engineering. Technical Report LIFL-2004-10, LIFL (2004)
10. Falleri, J.R., Huchard, M., Nebut, C.: Towards a traceability framework for model transformations in kermeta. In: ECMDA-TW Workshop. (2006)

Traceability and Provenance Issues in Global Model Management

Mikaël Barbero, Marcos Didonet Del Fabro, Jean Bézivin

ATLAS Group, INRIA & LINA
University of Nantes, Nantes, France
{mikael.barbero, marcos.didonet-del-fabro, jean.bezivin}@univ-nantes.fr

Abstract. Establishing and using traceability and provenance between different models is a very important issue in complex software systems. Traceability in the small (local traceability) handles the trace information between model elements. Traceability in the large (global traceability) handles traceability information between models as a whole. Current solutions are more or less ad-hoc approaches that mix local and global traceability. In this paper, we present global model management as a generic solution to support traceability in the large. Based on previous experience on traceability in the small using model weaving, we propose a novel approach that uses traceability megamodels. The traceability megamodels are extensible, thus supporting different kinds of traceability. Consequently, weaving models and megamodels are two complementary approaches to deploy a generic and complete traceability solution.

1 Introduction

Complex systems are represented by several kinds of models, for instance state chart models, UML models, transformation models, weaving models, and many others. These models are not isolated entities, but they are linked through different kinds of abstract relations. For instance, consider a service oriented architecture, where several services are executed to produce a particular output [7]. It may be necessary to know the sources that were used to generate this output, for example, to verify if the process is correctly executed, or to check if the sources are reliable. This scenario is typical in data provenance applications [10], where it is necessary to know which elements of a set of data sources were used to generate a target data source.

Consider another example of a bridge between two concrete syntaxes, KM3 [6] and SQL-DDL (Data Definition Language), having a MDE platform such as AMMA [1] as bridging platform. Several operations are executed during this process: the injection of the SQL-DDL file into the MDE platform, the production of the KM3 and SQL-DDL metamodels, the transformation of a KM3 model into a SQL-DDL model, and the extraction of the SQL-DDL model into its concrete syntax. To be able to trace back from one representation to another, it is necessary to record this chain of relations.

Traceability and provenance handling consists of storing information that enables to reconstruct these chains of operations. There are several issues that must be considered in traceability and provenance scenarios. First, it is necessary to know how to define fine-grained relationships between the elements of the models. For instance, to know which KM3 element was used to generate a target SQL-DDL element. In typical data provenance applications [10], the source and target models are annotated (or decorated) with the provenance information (we do not take into account issues such as security or privacy of the data). Despite being simple, these approaches have a major drawback: the source and target models are polluted with additional information that is not part of the model semantics. It is also necessary to modify the related query language to be able to read these traceability annotations. In addition, the complexity of the annotations may considerably increase when the number of relationships between models is large. The overhead of this solution may be too expensive. Moreover, it requires to adapt each metamodel to take in account the traceability issues. We rather think it should be profitable to have a loosely coupled solution. This solution would be sufficiently generic to be applied to any use cases but also extensible to be refined. Lastly, this approach would distinguish traceability of models elements from traceability of models, to allow different navigation levels: microscopic (model elements) and macroscopic (model).

The traceability and provenance between the elements of different models is called *traceability in the small*. We propose using weaving models to support traceability in the small. Weaving has already proved to be useful to represent relationships between different models in different application scenarios [2] [4]. A weaving model stores traceability information that is loosely coupled with the related models. This means the traced models are not modified or updated with extra traceability information. This enables to have a clear separation of the traced models from the traceability weaving model, and to have a unified view to handle all of these models. In addition, the weaving model conforms to a weaving metamodel that is extensible, thus supporting different kinds of traceability links. It is possible to create a hierarchy of metamodel extensions. This is very important because of the heterogeneity of the traceability scenarios.

However, fine-grained relationships between model elements are not enough to give a global view of the provenance of the data, because the weaving models contains double undirected links. For instance, in addition to discovering which elements of a given model are the origins of a target model element, it is necessary to have a global view of all the models involved in the traceability process.

There are different approaches to support traceability in the large. The work from [7] [9] proposes a provenance metamodel that enables expressing at the same time traceability between model elements and models. The main drawback is that it does not provide a clear separation between global and local relationships. Moreover, those relationships do not have the same semantic most of the time. Thus, they should be represented by different models. In requirements traceability [8], several traceability metamodels are defined to store traceability information in different abstraction levels. For instance, a decision maker uses more general provenance and traceability information, while a developer uses traceability information in a smaller and more specific context.

We propose using global model management (GMM) to store global traceability relationships. It is a complementary solution to model weaving. We propose using *megamodels* to handle the global traceability relationships. The concept of megamodel was initially proposed in [3]. A megamodel contains relationships between models, for instance transformation models, weaving models, UML models, or metamodels. We call it *traceability in the large*. Similarly to weaving models, a megamodel conforms to an extensible metamodel. This enables to store different kinds of traceability information. In addition, there is a clear separation between traceability in the large and traceability in the small.

To summarize, the contributions of this paper are the following. First, we propose the use of weaving models to store traceability relationships between model elements. We validate our approach by presenting a use case. Second, based on our experiences, we raise a set of issues that must be supported in a global traceability solution. We present a complete and generic proposal by using megamodels.

This paper is organized as follows. Section 2 presents a typical use case of traceability in the small, which is the traceability of model transformations. We show how weaving models are used to store the execution trace of ATL transformations. Section 3 presents issues of traceability in the large. Section 4 concludes.

2 Traceability in the Small

Traceability in the small consists of storing any relevant traceability information at the model element level. One typical application of traceability in the small is to store the execution trace of model transformations [5], i.e., to support traceability of model transformations. Based on previous work [5], this section presents an application scenario that uses weaving models to store the execution trace of ATL transformations.

2.1 Traceability of model transformations

A model transformation takes a set of models as input and produces a set of models as output. The elements of the input models are visited and then transformed into elements of the output models. After the execution of the transformation, it is necessary to discover which set of elements of the source models were visited and transformed into a set of target model elements, and which transformation rule was executed.

We present in Figure 1 a concise example of a transformation between two models. Although being very simple, this example clearly presents the problem of traceability of model transformations. The input model contains information about books (*Book*). It conforms to the book metamodel *MMb*¹. *MMb* has one class *Book*, which has attributes *title*, *author*, *year* (the content of the columns is self-explanatory). The

¹ The metamodels from this use case conform to the KM3 metamodel. KM3 is formed by classes, which are formed by attributes and references [6].

output model contains information about generic publications (*Publication*). It conforms to a publication metamodel *MMp*. *MMp* has two classes: *Publication*, which has attributes *title*, *authors*, *pubYear* and reference *authors* [multiple cardinality]; *Author*, with attribute *name*. The transformation *Mt* is an ATL transformation; thus it conforms to the ATL metamodel (denoted by *MMt*). A part of the code of the ATL transformation is shown in the right side of Figure 1. For every *Book* of the source model, the transformation creates a new *Publication* in the target model, and it assigns the values of the source attributes to the target attributes.

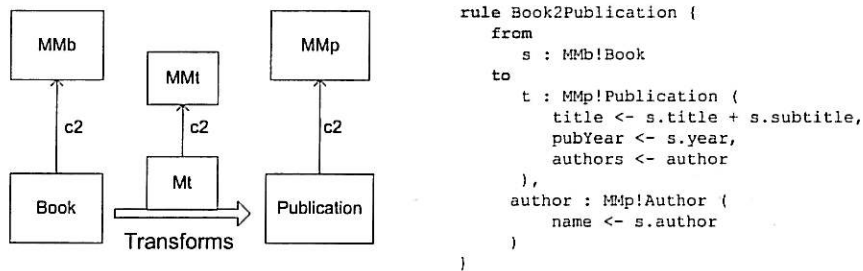


Fig. 1. Book to publication transformation

We illustrate the traceability between model elements in Figure 2. It shows one model element from the source model (*001 : Book*), and two elements from the target model (*002 : Publication* and *003 : Author*). The traceability information is represented by the lines between the model elements.

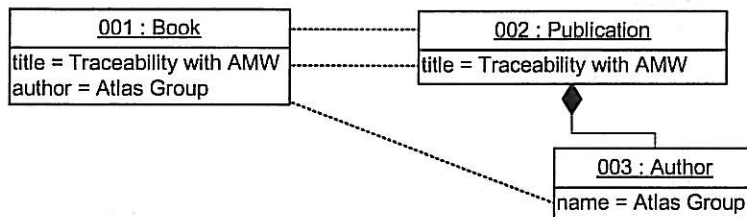


Fig. 2 Traceability between two model elements

Without this information, it is not possible to directly discover which elements of the source model are used to generate author *003*. The model transformation is created using the metamodel elements, and it is executed over the model elements. The relationships between the input and the output model elements (as well as the transformation rules), are accessible only in the moment of its execution. Consequently, without any traceability information, it would be necessary to apply an inverse procedure to transform the *Author* class into the *author* attribute, and to compare the result with the source model elements. These relationships must be saved to be able to exploit the execution trace information afterwards.

A weaving model can be used to capture this traceability information. This weaving model conforms to an extension to the core weaving metamodel. The traceability metamodel extension is depicted in Figure 3. The central element of this

extension is the *TraceLink* element. Every time a transformation visits a source element (e.g., the *001 : Book* in our case), it creates a new *TraceLink* in the weaving model. The reference *sourceElements* refers to the source elements (the *title* and *author* values, not the metaelements). The reference *targetElements* refers to the generated target elements (the multiple cardinality enables having more than one target element). The attribute *ruleName* has the name of the rule that is executed (e.g., *Book2Publication*). This attribute enables to keep a trace to the transformation rule, not only to the source elements.

The class *TraceLinkEnd* represents the source and target elements. The reference *element* (from the core weaving metamodel) refers to class *ElementRef*. This element is a proxy to the real linked elements. It saves an identifier that enables to uniquely identify the source or target model elements. The format of the identifier is specified by the annotation `--@wmodelRefType` (e.g., XMI IDs, XPointers, etc.).

```

class TraceLink extends WLink{
    attribute ruleName : String;
    -- @subsets end
    reference sourceElements[*] ordered container : WLinkEnd;
    -- @subsets end
    reference targetElements[*] ordered container : WLinkEnd;
}
class TraceLinkEnd extends WLinkEnd {
}
-- @wmodelRefType TraceModelRef
class ElementRef extends WElementRef {
}

```

Fig. 3. Traceability metamodel extension in KM3

However, the original model transformation *Book2Publication* does not specify how to create the traceability weaving model, only how to transform a *Book* into a *Publication*. Hence, the original transformation is modified into *Mt'*. *Mt'* has additional rules to create the elements of the traceability weaving model. The new setup is shown in Figure 4: the modified transformation *Mt'* takes the *Book* model as input and produces a *Publication* model and a traceability weaving model *Mw* as output (the metamodels are omitted for better visualization). The weaving model has the traceability links between *Book* and *Publication*.

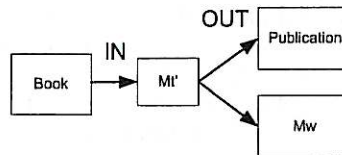


Fig. 4. *Mt'* generates a *Publication* model and a weaving model

To summarize, weaving models are an appropriate solution to provide traceability of model transformations. This can be extended to be used in different applications of traceability in the small, i.e., traceability between model elements. The traceability metamodel extension enables to create traceability links (domain specific links). The weaving model is automatically created when the ATL transformation is executed. The traceability weaving model can be visualized and modified on the AMW

prototype, without any modification on the source code. This shows the advantage of developing the prototype using a generic and reflective API. The traceability use case is available for download at (<http://www.eclipse.org/gmt/amw/usecases/traceability/>). This page contains a fully implemented example, with general documentation, a *HowTo* and the sources.

This example has been chosen to illustrate the characteristics of traceability in the small. We do not suggest that all such traceability in the small is related to model transformation. There are many other situations. Our purpose was mainly illustrative in this section.

3 Traceability in the Large

In the previous section, we introduced loosely coupled traceability in the small concepts. Despite the advantages compared to tightly coupled traceability, i.e., not polluting models², its main mechanism does not allow to follow traceability links because loosely coupled traceability relies on a double indirection linking. Thus, it is not possible to get the provenance of a target model element from itself as it does not know its trace model, or set of trace models. This principle is illustrated on Figure 5.

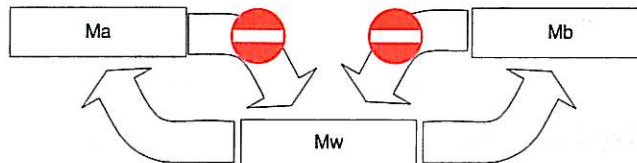


Fig. 5. Double indirection in weaving models and its limitations

Figure 5 shows that from the *Mw* model, we are capable of retrieving which elements of *Ma* and which elements of *Mb* are linked. The use of model *Mw* prevents from polluting *Ma* and *Mb* with any traceability metadata. This means *Ma* and *Mb* are not aware that they are linked by a weaving model. Thus, it is not possible to follow a link from *Ma* or *Mb* to the other model (depicted by the forbidden sign).

Moreover, the purpose of the traceability model *Mw* is to link model elements with traceability semantics. The link between the models as a whole has a different nature (and maybe semantics as well). Thus, the traceability model should not contain references to woven models. Current implementation keeps references to its woven models. This is an implementation trade-off but it does not completely separate global (between models) and local (between model elements) relationships.

We see that model weaving cannot answer the general goals for traceability or provenance. The only alternative is to pollute the models themselves with direct traceability links. This solution itself does not allow managing multiple traceability chains involving common models.

² As stated before, polluting a model is the action of adding information to this model unrelated with its original semantic.

At this point, the conclusion seems thus quite pessimistic. Should we give up using model engineering to provide a uniform, general and scalable solution to traceability and provenance problems? We decided to have a second try and to investigate how model weaving techniques could be complemented by megamodeling techniques to offer a complete solution. The result was positive and this proposal is the main contribution of the present paper.

We propose to see this issue of global relationship representation as a Global Model Management one. Global Model Management is about considering relationships between models themselves as a model. Thus, there is a metamodel of those models and every tool defined for modeling in the small are also applicable for modeling in the large: model transformation, model weaving, etc. We call this kind of model a megamodel.

A possible metamodel of a megamodel for traceability would be the one depicted on Figure 6. On this picture we can see that a *TraceModel* links one more source models to one or more target models. The *Model* class contains opposite references.

```

package TraceMegamodel {
    class Model {
        reference conformsTo : ReferenceModel;
        reference sourceOf[*] : TraceModel oppositeof sourceModels;
        reference targetOf[*] : TraceModel oppositeof targetModels;
    }
    class ReferenceModel extends Model {}
    class TraceModel {
        reference sourceModels[1-*] : Model oppositeof sourceOf;
        reference targetModels[1-*] : Model oppositeof targetOf;
    }
}

```

Fig. 6. A metamodel of a megamodel (KM3 notation)

Thus, with a megamodel conforming to this metamodel, it is possible to navigate from source model to trace model, from trace model to source and/or target models, from target model to trace model. Moreover, this solution does not pollute source, target, or trace models.

In the previous section, we introduced the metamodel extension mechanism, because there is more than one traceability metamodel depending on the desired granularity and purposes. In the same way, megamodels' metamodels are not unique. We have to provide a core metamodel with extensibility mechanism. Then, this metamodel extension will allow more or less navigation facilities, coarse or fine grained relationships between traced models for microscopic or macroscopic point of view.

4 Conclusion

In this paper we introduce a general and original proposal to traceability issues in the context of Global Model Management. Our proposal provides a clear separation of concerns between traceability in the small and traceability in the large. Traceability in

the small issues are addressed by model weaving, whereas traceability in the large issues are addressed by megamodeling. Model weaving has allowed not polluting traced model elements. In addition, megamodeling has allowed not polluting weaving models.

Apart from this separation of concerns, there are many traceability scenarios that can not be addressed by a unique weaving metamodel and a unique megamodel metamodel. By defining a simple core weaving and megamodel metamodels that can be extended, our solution fits several traceability requirements. The symmetry between extensibility for model weaving and megamodeling comes as a nice regular property of our proposal. This enables to query a complete traceability chain without polluting trace models and give an answer to a subset of data provenance requirements.

The study of different traceability use cases using our global model management approach and especially requirements traceability represents area of future work, but we are quite confident that the original idea presented in this paper is implementable with the existing prototype of AMW and the AMMA platform with high generality and low overhead.

Acknowledgements

This work has been partially supported by the ModelPlex European integrated project FP6-IP 034081 (Modeling Solutions for Complex Systems).

References

- [1] Allilaire, F, Bézivin, J, Didonet Del Fabro, M, Jouault, F, Touzet, D, Valduriez, P. AMMA : vers une plate-forme générique d'ingénierie des modèles. *Génie Logiciel*(73):8-15. 2005
- [2] AMW Use cases : <http://www.eclipse.org/gmt/amw/usecases/>
- [3] Bézivin, J, Jouault, F, Valduriez, P. On the Need for Megamodels. In proc. of OOPSLA & GPCE Workshop on Best Practices for Model Driven Software Development, Vancouver, Canada, 2004
- [4] Didonet Del Fabro, M, Bézivin, J, Valduriez, P. Model-Driven Tool Interoperability: An Application in Bug Tracking. In proc. of ODBASE'06, LNCS 4275, edited by R. Meersman and Z. Tari et al. Springer-Verlag Berlin Heidelberg 2006, pp 863-881. 2006
- [5] Jouault, F. Loosely Coupled Traceability for ATL. In proc. of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany. 2005
- [6] Jouault, F, Bézivin, J. KM3: a DSL for Metamodel Specification. In proc. of 8th FMOODS, LNCS 4037, Bologna, Italy, 2006, pp 171-185
- [7] Moreau, L, Chapman, S, Schreiber, A, Hempel, R, Rana, O, Varga, L, Cortes, U, Willmott, S. Provenance-based trust for grid computing. Electronics and Computer Science. University of Southampton. Position paper. 2004
- [8] Ramesh, B, Jarke, M. Toward Reference Models for Requirements Traceability. *IEEE Transactions on Software Engineering*. V27, Issue 1 (January 2001), pp 58-93

- [9] Szomszor, M, Moreau, L. Recording and reasoning over data provenance in web and grid services. In proc. of ODBASE'03, volume 2888 of Lecture Notes in Computer Science, pages 603-620, Catania, Sicily, Italy, November 2003
- [10] Velegrakis, Y, Miller, R J, Mylopoulos, J. Representing and Querying Data Transformations. In proc. of International Conference on Data Engineering (ICDE), pp 81-92, April 2005

10

Traceability-based Change Management in Operational Mappings

Ivan Kurtev¹, Matthijs Dee¹, Arda Goknil¹, Klaas van den Berg¹

¹ Software Engineering Group, University of Twente
7500 AE Enschede, the Netherlands
{kurtev, K.G.van.den.Berg, goknila}@ewi.utwente.nl
m.r.dee@student.utwente.nl

Abstract. This paper describes an approach for the analysis of changes in model transformations in the Model Driven Architecture (MDA). Models should be amenable to changes in user requirements and technological platforms. Impact analysis of changes can be based on traceability of model elements. We propose a model for generating trace links between model elements and study scenarios for changes in source models and how to identify the impacted elements in the target model.

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. Software architectures must be designed such that they can evolve to cope with these changes. The Model Driven Engineering (MDE) approach aims at providing stable models amenable to changes [14]. An analysis of the impact of changes is necessary for a cost effective software development [3]. The number of affected modules or elements is a response measure for the quality attribute *modifiability* in software architectural design [4]. Such analysis can be based on dependency traces between elements in the architectural design and other software artifacts.

In MDE, models are manipulated via model transformations. Transformations are usually sequentially applied until a model with enough details is obtained. A change in one of the source models causes changes in all the models obtained as products in the transformation chain. There are two options for performing these changes: (1) executing the transformations again on the whole modified model and (2) propagating the changes incrementally by transforming only the changed source elements. In the latter case we need an incremental model transformation mechanism. It is clear that in the case of large models the incremental transformation approach may be more efficient.

In order to perform change impact analysis we need to trace changes in a source model element to required changes in the target model elements. We utilize trace information created during the transformation execution. It provides a set of traces

that relate source and target elements created by a given transformation rule. The necessity of such traces points to the requirement for an important quality property of model transformation languages and execution engines: *traceability*.

Traceability is defined as the degree to which a relationship can be established between two or more products of the development process [10]. In the context of model transformations the products are models and their model elements. Traceability is an optional requirement in the QVT Request for Proposals (RFP) issued by OMG [15]. QVT specification describes three model transformation languages: Relations, Core, and Operational Mappings. In the Relations and Operational Mappings languages, traces are created automatically and remain transparent to the user. In the Core Language, a trace class is specified explicitly for each transformation mapping.

In this paper we study the possibility for using incremental model transformations written in the QVT language Operational Mappings. We use the transformation engine provided by Borland Together Architect 2006 for Eclipse to execute transformations and to experiment with the trace information. We evaluate the possibility to use the traces generated by Together Architect as a side product of a transformation execution. We classify change cases and analyze them in the context of incremental model transformations.

The paper is structured as follows. In Section 2, we describe our approach to generation of traces. Section 3 gives a conceptual framework for change impact analysis. Section 4 discusses possibilities and obstacles in implementing the identified changes. Section 5 describes related work and Section 6 concludes the paper.

2 Traceability in Operational Mappings Language

We generalize the concept of traceability by means of a traceability pattern (see Figure 1).

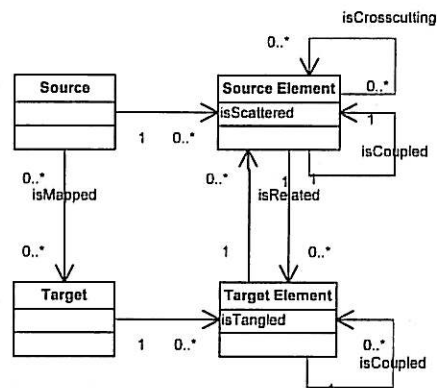


Figure 1. Traceability Pattern

In this pattern, we have dependency relations between elements in the source and elements in the target. We use here the general terms *source* and *target* to denote two consecutive levels. We distinguish between intra-level and inter-level dependency

relations. Intra-level relations denote couplings between elements at a certain level (model). Examples of intra-level relations are generalization between classes, aggregation, etc. Inter-level relations relate elements at different level of abstraction. For example, an element in a model is refined to a set of elements in another model at a lower level of abstraction. In this case we have a *refinement* inter-level relation. In the case of model transformations inter-level relations are the traces derived during the transformation of source elements to target elements. For the purpose of change impact analysis both types of dependencies should be taken into account.

We may distinguish several cases of mappings between source and target: 1-to-1, 1-to-many, many-to-1, and many-to-many. This can be represented in a dependency graph, as shown in Figure 2.

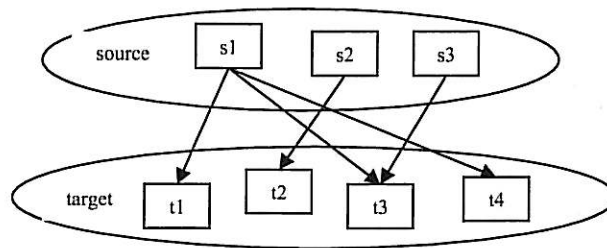


Figure 2. Mapping between elements at different levels of abstraction (s1, s2, s3 at source; t1, t2, t3 and t4 at target)

In general, a transformation language provides different ways for creating and maintaining the traces between source and target elements. Operational Mappings language uses automatic creation of the traces. They can be used during the transformation execution by invoking different forms of the *resolve* function. The QVT specification does not impose any constraints on the exact structure of the traces and their lifetime. This is considered an implementation specific issue.

The engine implemented in Together Architect provides an option for creating persistent traces, i.e. traces that are saved after the transformation execution. Keeping such traces in a form accessible to programs is essential for the change impact analysis. Unfortunately, Together Architect only provides a browsing mechanism to inspect the traces. They cannot be manipulated programmatically. This is the reason for implementing our own traceability mechanism that produces persistent traces in the form of a model.

In our approach traces are instances of a simple model. Every trace is associated to a rule and refers to the sets of source and target elements used by that rule. Formally traces have the following structure:

$$t([s1, \dots, sn], [t1, \dots, tm], r)$$

This is interpreted in the following way: trace t is derived from the execution of rule r on source model elements $s1, \dots, sn$ that results in the creation of target model elements $t1, \dots, tm$. Since a rule may match multiple tuples in the source model it is clear that multiple traces may exist per single transformation rule.

The set of traces form a model generated after the execution of a transformation. The generation is done by inserting code in every transformation rule in a given transformation definition. The automation of this process is beyond the scope of this paper. More information may be found in [12]. The model with traces conforms to a trace metamodel. It should be noted that this metamodel is different from the traceability pattern in Fig. 1. The trace metamodel describes in details the relations between source and target, and between source element and target element.

3 Traceability-based Change Impact Analysis

We present a classification of change cases and analyze them in terms of traces, rules to be executed, and elements affected in the source and target models. Due to a lack of space we give only three cases. The complete set of cases is available in [8].

3.1 Notation

In the remainder of this section models will be used to show the inter-level trace dependencies between source and target model elements. The legend for these models is shown in Figure 3.

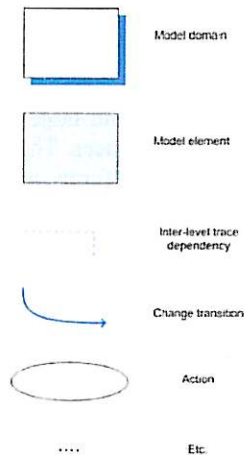


Figure 3 Visual notation used in the examples

We assume that models are sets of elements. We denote models in the following way:

Source model: $S = \{s_1, s_2, \dots, s_n\}$; Target model: $T = \{t_1, t_2, \dots, t_n\}$

The target model is generated by an execution of a transformation definition R that consists of transformation rules: $R = \{r_1, r_2, \dots, r_k\}$

During the execution of the transformation traces are created. These traces hold source and target model elements and the transformation rule which created the trace:

Set of traces: $I = \{i_1, i_2, \dots, i_1\}$; Trace: $i_1 = (s_1, t_1, r_1)$

We need to indicate changes being made to a model. In the following sections we limit ourselves to two change types: *update* and *delete*. In some cases we need to denote a change without specifying the type. We use the wildcard (*) in this case.

Change-types: $C = \{c_u, c_d, c_*\}$, where *u* stands for *update* and *d* for *delete*.

With these sets we can define a method that takes a change, a model, and the impacted elements, and creates the new desired model:

`change([change-type], [model], [element(s)]) = [model]'`

where:

[model] is a model, element(s) is a set of model elements affected by the change, and [change-type] is a change-type from C.

When applying a change to a source model, we need to keep the target model consistent with this change. This consistency is preserved with the implementation of the same change on the target model and thus the impacted element(s). The unidirectional transformation from S to T is indicated by *creates*. We assume that the target model is not changed in the meantime (e.g. edited manually) and all the changes are driven by changes in the source model.

3.2 Example Case: one-to-one Mappings

This case has two sub-cases:

- a single source model element is mapped to a single target model element with one transformation rule;
- multiple transformation rules create the target model element;

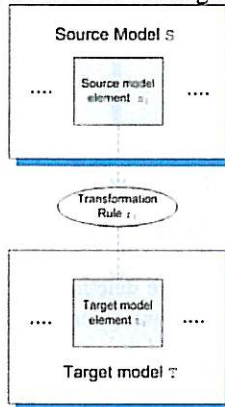


Figure 4 One to one inter-level trace dependency created with one transformation rule

Figure 4 shows a one-to-one trace example where the element s_1 is transformed to the element t_1 by executing rule r_1 : $r_1(s_1) = t_1$. The only trace created holds source element s_1 , target element t_1 and transformation rule r_1 .

The data contained in the source element s_1 is mapped to target element t_1 . This indicates the lowest level of granularity on which this information is used to create source elements' data.

When changing s_1 the impacted elements are found by following the trace leading to the target model element t_1 . The trace indicates the usage of the transformation rule r_1 . The change on s_1 must be propagated to the impacted target model element. The following facts are known:

$$S \text{ creates } T, i = (s_1, t_1, r_1), \text{ Change}(c., S, \{s_1\}) = S', \\ \text{Change}(c., T, \{t_1\}) = T'$$

The desired implementations of the change types are schematically shown in Figure 5 and Figure 6.

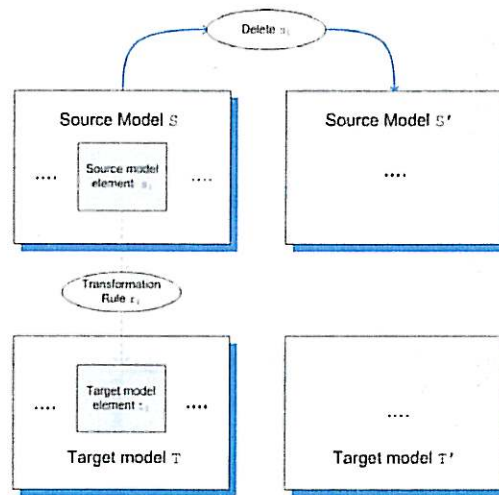


Figure 5 Delete change in one to one mapping

The change types infer two possible scenarios:

- *Delete the source element*

Deleting s_1 from S implies the deletion of t_1 from T . This results in the following change implementations, which creates the new source and target models:

$$\text{change}(c_d, S, \{s_1\}) = S \setminus \{s_1\}$$

$$\text{change}(c_d, T, \{t_1\}) = T \setminus \{t_1\}$$

- *Update the source element*

Updating s_1 from S implies the re-execution of transformation rule r_1 and replacing t_1 with the newly created target model element. This results in the following change implementation:

$$\text{change}(c_u, S, \{s_1\}) = S \setminus \{s_1\} \cup \{s_1'\} \text{ where } s_1' \text{ is manual input}$$

$$\text{change}(c_u, T, \{t_1\}) = T \setminus \{t_1\} \cup \{t_1'\} \text{ where } t_1' = r_1(s_1')$$

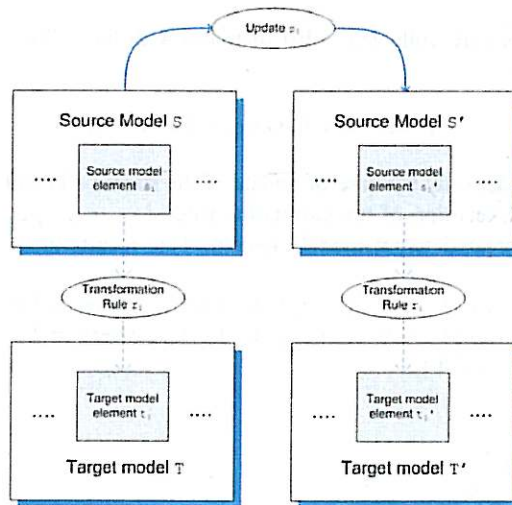


Figure 6 Update change in one-to-one mapping

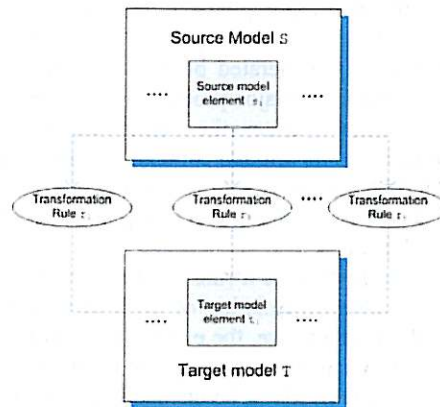


Figure 7 One to one inter-level trace dependencies created with multiple transformation rules

The previous example can be extended to a sub-case where the source element is mapped to one target element by using of multiple transformation rules. The difference between the two cases is the number of traces created during the execution of the transformation and the number of rules executed to create the target element. This is schematically shown in Figure 7.

In this example a trace is created for every transformation rule execution. We assume that there is a fixed order in which these transformation rules are executed (reflecting the imperative nature of Operational Mappings). Because of this assumption we get a sequential list of transformation rules $\{r_1, r_2, \dots, r_n\}$.

When the traces are collected, a list of traces with the following structure is created:

$$\{ (s_1, t_1, r_1), (s_1, t_1, r_2), \dots, (s_1, t_1, r_n) \}$$

Here we consider only the case of update of the source element. Updating s_1 from S implies the re-execution of transformation rules $\{r_1, r_2, \dots, r_n\}$ and replacing t_1 with the newly created target model element. This results in the following change implementation:

$$\begin{aligned} \text{change}(c_u, S, \{s_1\}) &= S \setminus \{s_1\} \cup \{s_1'\} \text{ where } s_1' \text{ is manual input} \\ \text{change}(c_u, T, \{t_1\}) &= T \setminus \{t_1\} \cup \{t_1'\} \text{ where } t_1' = r_1(s_1'); \\ &r_2(s_1'); \dots; r_n(s_1'); \end{aligned}$$

4 Discussion

In this paper we limit ourselves only to performing impact analysis. The next step is to implement the changes. We give an overview of possibilities and the potential obstacles.

In general, there are two ways to implement the required changes: generate a function that implements the change (a kind of "patch") and re-execute rules.

A patch function should be generated based on the logic of the transformation rules responsible for changes. A major problem is the imperative nature of operational mappings rules. It is not clear how the required functionality is extracted from a rule and how it is connected to its original context. We may hypothesize that using an imperative transformation language is not the best option for this scenario. The applicability of a language based on another paradigm (e.g. declarative, graph transformation-based) needs a further study.

The second option is to re-execute a rule. In the worst case this option may completely fail. In the current implementation of Borland together it is not possible to execute an arbitrary rule. Furthermore, the engine does not provide any customizable features. Apart from that, we also have a problem of granularity. Consider a case in which only an attribute value needs to be changed. This is done by executing of an assignment part of a rule. The smallest executable modules in Operational Mappings language are helpers and mappings. We cannot execute parts of a mapping. Instead, we need to execute the whole mapping. This, however, may lead to execution of other mappings invoked by the required one. There is no way to prevent this without knowing the internal implementation details of the transformation engine.

The current version of the approach to impact analysis has three major limitations:

- *Lack of intra-level reasoning.* When a model element is changed this may lead to changes in other elements in the same model due to the dependencies imposed by the language semantics. We call such dependencies intra-level dependencies. For example, if a class is deleted then its attributes must also be deleted. Another example is the effect of deleting a super class. This automatically changes the set of attributes of the specializations of that class. Both examples are related to the semantics of the modeling language being

used. The lack of standard way to define language semantics prevents us from performing generic intra-level reasoning. It should be done per every modeling language thus limiting the generality of the impact analysis framework.

- *Changes in guards are not considered.* Our tracing model keeps track on the relations between source and target model elements established during transformation execution. However, a change may also be caused by changing values that are checked by mapping guards. A source element may not be mapped and therefore no trace for it is available. Yet, this element may influence the evaluation of the guard in a mapping. Changing such an element will have no consequences since no trace is available for it. The only way to overcome this limitation is to enhance the trace model by including also the elements participating in the guard expressions.
- *Additions of elements are not handled.* Addition is problematic due to the same reason mentioned in the previous case: no trace is available for the newly added element. To handle the change, we need to identify the applicable mappings. However, due to the imperative nature of the language, such a mapping may be invoked in the context of another one and may invoke other mappings. Again, identifying the functionality for the change implementation is problematic.

5 Related Work

The topic of incremental model transformations is studied in [9] and [11]. Hearnden et al. [9] studies how continuous handling of changes in a source model may be done with the Tefkat language [13]. Their analysis is based on a detailed knowledge of the execution semantics of the language. The implementation of the changes extends the engine with additional structures for keeping intermediate execution context information. In this paper we study another language with different semantics. It was not possible to come with implementation of changes since we could not extend the transformation engine for Operational Mappings.

The authors of [11] propose writing model transformations in a style that is based on anticipating changes. Transformations are event-driven. Events are adding and deleting model elements. Events trigger transformation rules. This paradigm is different from the current model transformation languages that are not designed to anticipate changes in the models.

In [1], a number of events and change actions have been defined as part of an operational semantics of traceability. These events and change actions could be the start of a change impact analysis as described in this paper.

In [16], the generation of traceability links is discussed, especially between requirements and the object model, and between requirements. This corresponds to the inter-level dependencies and the intra-level dependencies as described in this paper. Similarly in [5], traceability links are retrieved between UML and target models including one-to-many relations. In the current paper, we focused on generated trace relations as part of QVT transformations. An event-based approach to traceability is

described in [6]. In this approach, change is handled by means of event notification and propagation of changes using traces between artifacts.

6 Conclusion

In this paper, we described an approach for the analysis of change in model transformations. We defined a traceability model for generating traces between source and target model elements. We analyzed the change impact in case of modifications of the source model. Several change scenarios were analyzed.

The specification of transformation rules and the tracing information of the execution of these rules can be used to generate dependency graphs at model level and at metamodel level. These dependency graphs are helpful in identifying which rule applications need to be re-executed in which order when there are changes to the source model. These are the key issues to implement incremental model transformations.

Our approach needs further elaboration and has to be validated in empirical case studies. Moreover, the derivation of dependencies and its analysis should be supported by tools in order to scale to industrial projects.

Acknowledgement

This work is performed in the context of AOSD-Europe Project IST-2-004349-NoE [2], the ESI Project Darwin [7] and the Jacquard/NWO QuadREAD Project.

References

1. Aizenbud-Reshef, N., Paige, R.F., Rubin, J., Shaham-Gafni, Y. and Kolovos, D.S. (2005). Operational Semantics for Traceability. In ECMDA-FA Traceability Workshop, Nuremberg, Germany
2. AOSD-Europe (2005). *AOSD Ontology 1.0 - Public Ontology of Aspect-Oriented*. Retrieved May, 2005, from <http://www.aosd-europe.net/documents/d9Ont.pdf>
3. Arnold, R. S., & Bohner, S. A. (1993). Impact analysis - towards a framework for comparison. Paper presented at the Conference on Software Maintenance.
4. Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice* (2nd ed.). Boston: Addison-Wesley
5. Berg, K. van den & Conejero, J. (2005). Disentangling crosscutting in AOSD: a conceptual framework, in *Second Edition of European Interactive Workshop on Aspects in Software*, Brussels, Belgium
6. Cleland-Huang, J., C.K. Chang, and M. Christensen (2003). Event-based traceability for managing evolutionary change. *IEEE Transactions on Software Engineering* 29(9) pp. 796-810
7. Darwin (2005). *Designing Highly Evolvable System Architectures*. Retrieved March 13, 2006 from <http://www.esi.nl/site/projects/darwin.html>

8. Dee, M. (2007). Traceability-based change impact analysis in MDA. MSc Thesis, University of Twente, Dept. Computer Science.
9. Hearnden, D., Lawley, M., Raymond, K.: Incremental Model Transformation for the Evolution of Model-Driven Systems. *MoDELS 2006*, pp. 321-335
10. IEEE (1990). IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. Institute of Electrical and Electronics Engineers, New York
11. Johann, S. and Egyed, A. "Instant and Incremental Transformation of Models," *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*, Linz, Austria, September 2004, pp. 362-365
12. Kurtev, I., Berg, K. v. d., & Jouault, F. (2006). Evaluation of rule-based modularization in model transformation languages illustrated with ATL. Paper presented at the 21st Annual ACM Symposium on Applied Computing, Bourgogne University, Dijon, France.
13. Lawley, M., Steel, J. Practical Declarative Model Transformation with Tefkat. *MoDELS Satellite Events 2005*. pp. 139-150
14. MDA (2003). MDA Guide Version 1.0.1, document number omg/2003-06-01
15. OMG (2002). Request for Proposal: MOF 2.0 Query/Views/Transformations RFP, OMG document ad/2002-04-10
16. Spanoudakis G., Zisman A., Perez-Minana E., Krause P. (2004). Rule-Based Generation of Requirements Traceability Relations , *Journal of Systems and Software*, Vol 72(2), pp 105-127

10