Proceedings of the $6^{th}$ ECMFA Traceability Workshop
(ECMFA-TW)

June 15$^{th}$ 2010, Paris, France

This workshop was organized in collaboration with the European Conference on
modeling foundations and applications (ECMFA 2010).

ACM International Conference Proceedings Series

ACM Press

Editors:
Jon Oldevik, Goran K. Olsen - SINTEF
Dimitrios S. Kolovos – University of York
# Table of contents

Editorial  
*J.Oldevik, G.K.Olsen, D.S.Kolovos*

**Keynote - Traceability in Model-Driven Safety Critical Software Engineering**  
*R.F.Paige*  

A Generic Traceability Framework for Facet-based Traceability Data Extraction in Model driven Software Development  
*B.Grammel, S.Kastenholz*

Tooling for Full Traceability of Non Functional Requirements within Model Driven Development  
*A.Yrjönen, J.Merillinna*

A State-Based Approach to Traceability Maintenance  
*N.Drivalos-Matragkas, D.S.Kolovos, R.F.Paige, K.J.Fernandes*

Resolving Feature Dependency Implementations Inconsistencies during Product Derivation  
*Saad Bin Abid*

Tool Support for Generation and Validation of Traces between Requirements and Architecture  
*A.Goknil, I.Kurtev, K.van den Berg*

An Aspect-based Traceability Mechanism for Domain Specific Languages  
*R-H.Pfeiffer, A.Wasowski*

A Transformation Instance-Based Approach to Traceability  
*T.Levendovszky, D. Balasubramanian, K.Smyth, F.Sh, G.Karsai*
Programme committee

Stefan Van Baelen, K.U.Leuven, Belgium
Mariano Belaunde, France Telecom, France
Ivan Kurtev, Univ. of Twente, The Netherlands
Miguel de Miguel, Universidad Politecnica de Madrid, Spain
Tor Neple, Norse Solutions, Norway
Richard Paige, Univ. of York, UK
Tom Ritter, Fraunhofer FOKUS, Germany
Andreas Rummler, SAP Research, Germany
Mirko Seifert, Univ. of Dresden
Ståle Walderhaug, SINTEF, Norway
Introduction to the Workshop

This is the sixth Traceability Workshop organized in the context of the ECMFA (previously ECMDA) conference series. For this year’s workshop we have accepted 7 papers after detailed review and revision. The focus of the papers spanned from the quite theoretical to implemented solutions for practical problems.

Traceability in Model Based Engineering (MBE) is an important aspect, and involves the linking of different development artifacts based on how they relates to each other. The links may have rich semantics that enables complex analysis and tooling, which provides value for the end users, e.g. help verifying that a requirement has been implemented or determining the source transformation of a generated UML Class.

There are still many challenges related to traceability in MBE, ranging from how and when to establish links, how to maintain the links, synchronization of the links and the granularity of the trace links. Many of these issues are targeted in this year’s proceedings, which are summarized below.

Levendovszky et al. present a chain of tools that provides traceability from transformations. In the paper “Resolving Feature Dependency Implementations Inconsistencies during Product Derivation” Abid uses traceability to check the dependencies between cross-cutting features in Software Product Lines (SPL) for consistency. Drivalos-Matragkas et al. present a state-based approach to traceability maintenance and also includes a summary of general concerns and challenges connected to trace-link maintenance. “A Generic Traceability Framework for Facet-based Traceability Data Extraction in Model-driven Software Development” by Grammel et al. presents a framework for recording and data extraction of traceability information in model driven software development.


We trust that the workshop papers, and the presentation of these, act as catalyst for constructive discussion both on the theoretical and practical aspects of traceability in Model Based Engineering. The number of submitted and presented papers indicates that the topic of traceability still is a focus within the community. From last year’s workshop the message taken away by the organizing committee was that industrial adoption of traceability techniques and theories was still hindered by lack of mature tools and standardization. During this year’s workshop we would like to revisit these topics, among others, to see if there has been improvement on these areas. We are also continuing the Tool Demonstration Track from last year, which will feature several tool demonstrations. The workshop will also feature a keynote by Professor Richard F. Paige from the University of York.

With this we wish you all a fruitful workshop.

Categories and Subject Descriptors
D.2.10 [Software Engineering]: Design

General Terms
Traceability, Model-Based Engineering

Acknowledgment

The 6th ECMFA Traceability Workshop has been made possible through the following projects:

MADES project, co-funded by the European Commission under the "Information Society Technologies" Seventh Framework Programme (2009-2012).

SiSaS - Scientific Software as a Service, Funded by SINTEF Norway (2009-2012)

ITEA Project MoSiS - Model-driven development of highly configurable embedded Software-intensive Systems, ITEA ip06035

ITEA Project VERDE - Validation-driven design for component-based architectures, ITEA ip08020

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2010 ACM 978-1-60558-993-0/10/06 ...$10.00
Traceability in Model-Driven Safety Critical Software Engineering

Richard F. Paige
Department of Computer Science
University of York
York, United Kingdom
paige@cs.york.ac.uk

ABSTRACT
Evidence-Based Software Engineering (EBSE) focuses on understanding and delivering software engineering practices, tools and techniques that qualitatively and quantitatively provide value. The principles of EBSE underpin safety critical software engineering practices: when we build a safety critical software system, we must, in parallel, deliver evidence that the steps we have taken and the artefacts that we build will lead to an acceptably safe system. In safety critical software engineering, traceability plays a vital role. This talk will explore some of the different applications and uses of traceability in this context, and will suggest ways in which Model-Driven Engineering can provide solutions (e.g., through standardised approaches for describing safety arguments and evidence), as well as new challenges.

Categories and Subject Descriptors
D.2.10 [Software Engineering]: Design

General Terms

Keywords
Traceability, Model-Driven Engineering, Safety Critical Systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ECMFA-TW ’10, June 15, 2010, Paris, France
Copyright 2010 ACM 978-1-60558-993-0/10/06 ...$10.00.
A Generic Traceability Framework for Facet-based Traceability Data Extraction in Model-driven Software Development

Birgit Grammel
SAP Research CEC Dresden
Chemnitzer Str. 48, 01187 Dresden, Germany
birgit.grammel@sap.com

Stefan Kastenholz
SAP Research CEC Dresden
Chemnitzer Str. 48, 01187 Dresden, Germany
stefan.kastenholz@sap.com

ABSTRACT

Traceability of artefacts induces the means of understanding the complexity of logical relations existing among artefacts, that are created during software development. In turn, this provides the necessary knowledge for reasoning about the quality of software. With the inception of Model-Driven Software Engineering, the advantage of generating traceability information automatically, eases the problem of creating and maintaining trace links, which is a labor intensive task, when done manually. Yet, there is still a wide range of open challenges in existing traceability solutions and a need to consolidate traceability domain knowledge. This paper proposes a generic traceability framework for augmenting arbitrary model transformation approaches with a traceability mechanism. Essentially, this augmentation is based on a domain-specific language for traceability, accounting for facet-based data extraction.

1. INTRODUCTION

In the IEEE Standard Glossary of Software Engineering Terminology [1] the notion of traceability is defined as: The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match. Traceability data in model-driven software development (MDSD) [16] can be understood as the runtime footprint of model transformations [5]. Essentially, trace links provide this kind of information by associating source and target model elements with respect to the execution of a certain model transformation. The corresponding models may conform to the same or different metamodels. Trace links have a manifold application domain [15, 5]:

• System analysis to understand system complexity by navigating via trace links along model transformation chains

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ECMFA-TW ’10, June 15, 2010, Paris, France
Copyright 2010 ACM 978-1-60558-993-0/10/06 ...$10.00.

According to [5, 17] model transformation approaches either generate trace links implicitly or explicitly. That is, in the former case, either provide an integrated support for traceability or in the latter one, rely on a developer to encode traceability as a regular output model. Nevertheless, these foreseen, orthogonal traceability solutions still entail a wide range of open challenges. One challenge is that the collected traceability data does not adhere to a unified traceability metamodel resp. data format, which aggravates tool interoperability and reasoning over traceability data. At the same time traceability data needs to bring up a certain expressiveness to account for the above mentioned traceability-specific scenarios, which is not necessarily amenable to a traceability metamodel claiming universality. Finding the sweet spot between these directives is a non trivial task [2]. Furthermore, there is yet room for improving the means to populate a traceability metamodel, requiring manual encoding for individual transformations, which is likely to be a cost and time intensive task.

In this paper, we propose a generic traceability framework for augmenting arbitrary model transformation approaches with a traceability mechanism. This generic traceability framework is based on a domain-specific language for traceability (Trace-DSL), presenting the formalization on integration conditions needed for implementing traceability. Essentially, this language agnostic Trace-DSL provides a unified traceability metamodel, yet on the other hand accounts for an adequate expressiveness of traceability data needed for the traceability-specific scenarios. To achieve this dual nature, the Trace-DSL is featured with an extensibility mechanism based on facets [14].

To demonstrate the feasibility of our framework, we chose a representative from each of the trace link generation classes, that is, explicit (e.g. oAW [12]) resp. implicit (e.g. QVT [11]) class. Respectively for these representatives, we advocate two possible augmentation mechanisms to achieve
traceability: a) Augmentation of the generator logic based on aspect-oriented programming [6], and b) Augmentation of the traceability data output via a model transformation. The content of the following sections is structured as follows. At first, we reflect challenges for traceability in MDSD and give an overview on related work. In Section 3 we extract our problem definition. Thereafter, in Section 4, an outline of the proposed approach is given, followed by an illustrative example in Section 5 for both augmentation mechanisms to achieve traceability. Finally, Section 6 gives an outlook on future work and Section 7 summarizes this paper.

2. CHALLENGES FOR TRACEABILITY IN MDSD

According to [5, 17] transformation approaches either generate trace links implicitly or explicitly, that is, in the former case, either provide an integrated support for traceability or in the latter one, rely on a developer to encode traceability as a regular output model.

The major advantage of implicit trace link generation (e.g. QVT [11], MOFScript [10]) is the fact, that no additional effort is necessary to obtain trace links between input and output models, as they are generated automatically in parallel to the actual model transformation. A disadvantage is, that the traceability metamodel is fixed and since most transformation approaches have differently defined metamodels, standardization among different approaches is aggravated. Essentially, the implicit generation allows less flexibility to control traceability data, that is [5]: a) The kind of information recorded, b) The granularity level of traceability data: (e.g. tracing only on file level instead of more concrete entities of the file body). Setting the granularity of trace links which may differ from one traceability scenario to another has been identified as a challenge by the Center of Excellence for Traceability [2]. When tracing all model element references, the number of trace links might become incomprehensible and hence less useful to the developer. Furthermore, it might be a performance issue when handling large and complex model transformations. c) The scope for which the information is recorded (e.g. tracing for specific rules, or subset of the source model). Not all model information might be allowed to be traced for security reasons, mandated for instance by customer needs.

Alternatively, as in the case of explicit trace link generation, it is possible to treat traceability as a regular output model of the transformation and incorporate additional transformation rules to generate it. (e.g. ATL [4], oAW [12]). The choice of metamodel is then completely at the discretion of the developer and does not depend on the transformation engine. Hence, the control over the traceability data modelling is given. The drawback however, is, that additional effort is required to add traceability-specific transformation rules, which may also pollute the implementation. As this task is generally done manually, it is likely to be error-prone and time consuming. Moreover, this effort is repeated for each transformation. An approach that partly solves these issues in ATL by automatically generating the traceability-specific transformation rules was proposed in [8].

Another challenge stated by [2] concerns the semantics of trace links. It is often necessary to distinguish between different kinds of links. For example, a link between a textual requirement and a model element has a different semantic, than a refinement relationship within a model. The required kind of links are often highly project dependent [9]. Fixing the kinds of semantic links, has the consequence of less flexibility for user-defined links that might be necessary to meet different project or company needs. On the other hand, since the choice of semantics, attributed to a link, is guided by the reasoning about what the user will perform with the link [15], not predefining the link semantics may result in failure of reasoning, due to misuse of the semantics.

3. PROBLEM DEFINITION

From the above dichotomy (cf. Section 2), we extract the main issues and challenges we wish to tackle with our approach as part of our requirements analysis:

- **Unification of traceability metamodels:** While the implicit trace link generation class provides an integrated traceability solution, which does not account for the adoption of traceability data, the explicit trace link generation class does. This consequential discrepancy results in the problem of not generally having traceability data conforming to a unified traceability metamodel (not even with regard to a generation class itself).

- **Extensibility of traceability metamodel:** Since the kind of traceability data to be collected is dependent on the actual traceability goal, (i.e. traceability scenario) [9], a unified traceability metamodel (in accordance with the above point) might not account for an adequate data expressiveness to conduct all traceability scenarios. Thus, an extensibility mechanism is required to deliver traceability scenario specific metamodels on the basis of a unified traceability metamodel. Essentially, this mechanism needs to hold for the specification of granularity and scope as well as the trace link semantics w.r.t. a certain traceability scenario, as argued in Section 2.

- **Minimization of efforts to achieve traceability:** Likely error-prone and time consuming factors of manual efforts to achieve traceability are to be reduced, as in the case of the explicit trace link generation class.

4. TRACABILITY APPROACH

Relying on current solutions on traceability (cf. Section 2), our approach aims at not implementing yet another transformation language, nevertheless, the aim is to consolidate benefits of implicit and explicit trace link generation and tackle their above-mentioned disadvantages resp. challenges. Essentially, our goal is to develop a generic traceability framework for augmenting arbitrary model transformation approaches with a traceability mechanism. A high-level architecture of our approach is depicted in Figure 1. Our framework is based on a generic traceability interface, which provides the connection point for arbitrary transformation engines. In this case the interface supplies the engineer with an API to connect his transformation engine to the trace engine, in terms of a corresponding connector. As a result, the transformation engine is featured with traceability functionality. Secondly, our approach is based on a domain-specific language for traceability (Trace-DSL), conditioning a formalized rule system on traceability data modelling, as
argued in [7] with the maxim to account for the traceability scenarios in Section 1. Essentially, the traceability data exchangeable between the generic interface and connectors resp. repositories conforms to the Trace-DSL, as indicated by the dashed arrows in Figure 1. We next describe the Trace-DSL (Section 4.1) and generic traceability interface (Section 4.2) in more detail.

### 4.1 Traceability Domain-specific Language

The Trace-DSL depicted in Figure 2 includes a root element, **TraceModel**, which contains several concepts, successively discussed in detail below. An **Artefact** represents any traceable product generated in the development process, such as a requirement or class, or a compound artefact, e.g. a method inside a class. Every artefact is unambiguously identified by a Universal Unique Identifier (UUID). A **TraceLink** is the abstraction for the transition from one artefact to another, such that an instance corresponds to a hyperedge linking two artefacts. A transition is always directed, therefore a from-to relation between artefacts is created by a trace link between source and target artefacts.

For assigning types to trace links, we recall from [7] our approach to follow a traceability scenario-driven link type derivation process with the maxim to define a minimal set of elementary links. Thereby, we focus on implicit links, referring to trace links between artefacts that arise due to a MDSD operation, e.g. a model transformation, as opposed to explicit links, referring to trace links between artefacts that are created manually (mostly), as per classification in [13].

Traceability, in our opinion, is the tracking of all changes possibly applied to models resp. model elements during a model transformation chain. In order to gather all possible changes we propose to break down a model transformation chain into its elementary operations (eo). For the resulting set of eo, we define for each type of eo, a certain link type between the corresponding source and target model elements. The link type derivation is based on the following classification, as shown in the table of Figure 3. According to [5] source-target relationships of model transformation approaches may be categorized into different classes: Either a new target model is created, and/or the transformation operates on an existing target model. The latter case may be split up into update transformations, where the existing target model -other than the source model- is updated, or in place transformations, where source and target are always the same model. Furthermore, an update transformation works destructively by removing model elements or by extension only, leaving existing model elements unaffected. Since the above classification states all possible relationships existing between source and target model elements, which may be interpreted as the result due to operations on model elements, we propose to derive the set of possible eo from the above classification. In addition, we base our definition on the well-known set of CRUD-actions, which seem promising as frame of reference to gather all possible eo on model elements [7]. Essentially, we identify four eo and define the corresponding trace links as follows:

- For a newly created target model, we define a **create link** between source model element/s and target model element/s.
- For an update (destructive or extension-only) as well as update in-place transformation, we define an **update link** between source model element/s and target model element/s.
- For a delete operation on a model element, we define a **delete link** as a special case of an update link.
- For a query (read) operation on a model, returning a subset of model elements from the source model, we define a **query link** between each source model element and returned target model element.

Therefore, **TraceLink** has four subtypes: **CreateTraceLink**, **UpdateTraceLink**, **QueryTraceLink** and **DeleteTraceLink**. The base entity TraceLink is abstract and therefore can not be instantiated, since the type system is expressive enough to account for the traceability of all possible model transformations [7].

Furthermore, to assign types to artefacts and trace links, we use the concept of facets [14], where the Trace-DSL assigns a set of facets (Facet) to every artefact and trace link. According to [14], the notion of facets has several slightly dif-
ferent definitions and is used under different names. Therefore, we will state the characteristics a facet owns in the context of our work: a) Facets own a model which consists of a hierarchy of facetted values and/or facets. (The difference between facet and facetted values lies in the level of abstraction. Facets are abstract, whereas facetted values are concrete as per choice of their implementation) b) Facets may form facet hierarchies. c) Every facet model contains a facetted value denoting an unknown facetted value.

We provide an example facet in Figure 4, namely a source code facet. This facet has two subfacets called TextFacade and JavaCodeFacet, as explained in the following. The TextFacade identifies code artefacts, either being of the nature text file or text block within such a file. To identify the location and name of a file resp. the startPosition and endPosition of a block in such a file, we introduce the two facetted values namely, TextFileFacade resp. TextBlockFacade, setting their attributes accordingly. For the purpose of identifying source code the usage of start and end position may not be sufficient. Therefore, a more advanced approach is provided by the JavaCodeFacade for tracing code written in the Java programming language. Hereby, we distinguish between facetted values referring to packages, classes, methods and attributes. Hence, it is possible to identify an artefact to be a Java method with a specific name, rather than by a string bounded by the start and end position of the TextBlockFacade.

The motivation for choosing facets is twofold. Since facets factorize inheritance hierarchies and thus, simplify inheritance hierarchies, we use this advantage for the sake of simplifying artefact- and link type hierarchies. Without the use of facets, a full model would multiply all classes, leading to the product of \( n_1n_2...n_f \) classes, where \( n_i \) denotes the number of instances holding for a facet, \( i \in N \), where \( N \) is the set of natural numbers, and \( f \) denotes the number of facets. As opposed to this, modelling with facets amounts to using \( n_1 + n_2 + ... + n_f \) classes. Secondly, facets account for an extensibility mechanisms for the type system of the Trace-DSL. Since facets can be varied independently, the extensibility of the traceability metamodel is achieved, aligned with our problem definition in Section 3. More precisely, the definition of traceability scenario specific metamodels through facet-based extensibility signifies the following adaptations: a) Selecting the required facets for a given traceability scenario and b) Configuration of Granularity and Scope.

The Configuration of the granularity level of traceability data implies the specification of which kind of artefacts respectively trace links with regard to a particular traceability scenario are to be traced. Thus, in the context of facets the granularity defines traceability scenario specific data by choosing facets resp. facetted values from the set of facets defined in a). For instance, regarding the previously mentioned code facets, the traceability of artefacts referring to text blocks might be too fine-grained, whereas artefacts referring to files might be sufficient. In this case, the trace engine would abstain from tracing artefacts with a TextBlockFacade.

The Configuration of traceability scope implies to constrain the traceability data in the sense of having a specific attribute-value combination. While the configuration of granularity solely checks for the existence of facets, the configuration of scope additionally examines a facet-specific property. For instance, regarding the TextFileFacet it might be necessary to trace only TextFiles of a certain name. In this case, the configuration of scope necessitates for the TextFileFacade’s attribute, name, to be set accordingly.

4.2 Generic Traceability Interface

The generic traceability interface (GTI) aims to allow the cooperation of arbitrary transformation and trace engines in order to collect traceability data. Consequently, we need to abstract from both the transformation engine performing model operations and the trace engine collecting traceability data. Figure 5 depicts the generic traceability interface. Interfaces regarding the transformation engine are summarized in the subpackage TransformationEngine, analogously, traceability engine concerns are grouped in the TraceabilityEngine subpackage. Common structures and behavior are contained in the Core package. In the following we will discuss every interface in detail.

Core: The core provides essential interfaces used by the transformation and trace engine. TraceLinks are characterized by a type, referring to the four TraceLink subclasses in the Trace-DSL. The relationship is modeled as an association to an enumeration. We disregarded an alternative subclassing-realization due to the fact that the types of trace links are fixed to a number of four, as specified in the Trace-DSL. Furthermore, TraceLinks aggregate a set of source and target artefacts while establishing a directed relationship among them. The only obligation an artefact has is to encapsulate an unambiguous reference, denoted as Universal Unique Identifier (UUID). A UUID is split up into a Unique Resource Identifier (URI) and a fragment. The URI is used to uniquely identify resources. This is an open-ended concept, since an URI may refer to a file in the file system, a web page on the internet or any other kind of data. A fragment on the other hand uniquely identifies objects within a resource. The usage of UUIDs solely depends on the implementation of the generic traceability interface. It is important that the connector resp. repository implementation define a mapping between the identification format used by the transformation resp. trace engine to the implementation of the UUID interface. Since facets (defining the artefact type) encapsulate any conceivable view on artefacts to allow for a flexible artefact type definition, the interface IFacet is kept as generic as possible, yet we require a facet to provide its name and value. Essentially, the Core package realizes the left part (including Facet) of the Trace-DSL, depicted in Figure 2.

Traceability Engine: In the TraceabilityEngine subpackage all interfaces relevant to the traceability engine are summarized. The central interface is the TraceEngineManager, providing access to registered repositories and configurations. Repositories implement the IRepository interface, which constitutes the connection point for arbitrary traceability engines to the generic interface. IRepository owns a name which serves as an identifier. Additionally, they own the methods connect() and disconnect(), which hold for possible database technologies. The most fundamental method is storeTraceLink(), which persists a trace link, its source and target artefacts and all contained facets. Finally, the exception handler methods are used for the configuration of traceability data, which essentially is realized by IConfiguration. IConfiguration has a validation method taking a TraceLink as input. Within the method the traceability data is checked whether the trace link and artefacts are
within the correct scope and granularity level.

Transformation Engine: The TransformationEngine subpackage contains all interfaces relevant to the transformation engine and provides the connection point for arbitrary transformation engines. The central interface is the TransEngineManager, which provides access to registered connectors and facet factories. Connectors implement the IConnector interface, which constitutes the connection point for arbitrary transformation engine to the generic interface. The generic interface expects the transformation engine to invoke the executeModelOperation() method, whenever a model transformation is performed. Thereby, the connector has to gather traceability data and create a corresponding TraceLink, which is passed with this method. Finally, the TransEngineManager includes IFacetFactories. Facet factories are facilities creating facet objects out of traceability data and/or transformation engine specific knowledge. For instance, the creation of the so-called TransformationEngineFacet -for tracing a specific transformation engine version- is completely traceability data independent. This is due to the fact that the current version of a transformation engine is available within its framework. On the contrary, the JavaCodeFacet factory (cf. Figure 4), solely applying to Java files is traceability data dependent. Additionally, there are facet factories which are decoupled from both factors, transformation engine and traceability data, e.g. a TransformationTimeFacet -for tracing the time of creation for a certain artefact w.r.t. a Gregorian time (date and time), or fiscal time (fiscal year and week) stamp- depending on a given Java Virtual Machine providing the correct time stamp.

In the following, we show the collaboration of the GTI interfaces in order to collect traceability data. First of all, the executeModelOperation() method is invoked by the IConnector, which triggers the tracing process, by creating according trace links.

After creating a trace link and its corresponding artefacts, facets have to be added. Therefore, the TransEngineManager is asked to deliver all registered IFacetFactories. Then facets are added to the traceability data by iterating over all source artefacts, target artefacts and finally, the currently processed trace link itself. In each iteration the createFacet() method of the IFacetFactory is invoked. In case the factory can be applied to the traceability data, a facet is returned and stored in the artefact respectively trace link.

For the registration of trace links to the repositories, first the TraceEngineManager is called, to get all traceability repositories, followed by the connection to the repositories. Thereafter, for ensuring that defined scope and granularity are respected, all registered configurations are requested from the TraceEngineManager. Then, the trace link is validated by every IConfiguration. Finally, the trace link is stored in the traceability repository and the connection is closed (only if the validation does not fail).

5. ILLUSTRATIVE EXAMPLE

To demonstrate our approach we provide an illustrative
example regarding the dichotomy of Section 2, thereby choosing a representative of each trace link generation class, namely oAW resp. QVT for the explicit resp. implicit generation class.

5.1 oAW connector

Our approach with regard to oAW is based on augmenting the oAW generator logic to obtain a traceability mechanism based on our Trace-DSL. Essentially, we chose aspect-oriented programming for the generator logic augmentation and demonstrate this with regard to the oAW model-to-text transformation language, Xpand [18].

The implementation of the GTI requires the adaption of a model transformation engine - in this case the oAW framework. Yet, the user perspective of the oAW framework should remain invariant to the use of the GTI, secondly we wish to minimize integration configuration efforts. Therefore, we adopt the concept of aspect-oriented programming by using AspectJ [3] to augment the generator logic of the oAW framework, such that, the framework itself stays untouched after the integration of the GTI.

Specifically, we base our approach on the following train of thoughts: In oAW model-to-text transformations are defined in terms of the Xpand language. The rules for transforming model elements are specified in so-called templates, i.e. defining instances of the Xpand language. At runtime, oAW instantiates an abstract syntax tree (AST), which contains Xpand statements. The execution of each model transformation necessitates for each statement, the invocation of the oAW internal $evaluateMethod()$ (among others), directing the AST instantiation. Thus, we use this method as a pointcut definition to implement the mapping of model transformations to traceability data by analyzing the AST at runtime, as follows. In particular, we only consider those statements that are relevant to collecting traceability data, as listed below. The derivation of traceability data out of Xpand statements is aligned with the semantics of the TraceDSL link types (cf. Section 4.1).

a) The $FILE$-statement invokes the generation of an output file from its body statement to the specified target, where the “expression” denotes the file name in the statement syntax:

```plaintext
<< FILE expression [outletName] >>
a sequence of statements
<< ENDFILE >>
```

Therefore, we derive CreateTraceLinks which connect the currently processed model element as source artefact and the actual file in the file system as the target artefact.

b) TEXT-statements results in textual information from the template being written in the output file, where a TEXT statement is any text, which is not surrounded by guillemets, e.g. public class. Thus, we populate CreateTraceLinks relating each text sequence from its template with its correspondence within the generated output file.

c) FOREACH-statements expand the body of the FOREACH block for each element of the target collection that results from the expression:

```plaintext
<< FOREACH expression AS variableName [ITERATOR iterName] [SEPARATOR expression] >>
a sequence of statements using variableName to access the current element of the iteration
<< ENDFOREACH >>
```

Thus, we define a CreateLink between the model element the expression refers to and the target collection that results from the expression. Additionally, we link the currently processed model element to the model element from which the output file was generated, with a QueryLink.

d) $EXPRESSION$-statement evaluates properties of the source model elements according to its metamodel. In order to capture this traceability relationship, we define CreateLinks between the source model element and the target text sequence denoted by the expression. A summary of the derivation is listed in Figure 6.

```
<table>
<thead>
<tr>
<th>Xpand2Statement</th>
<th>Link Type</th>
<th>Source Artefact</th>
<th>Target Artefact</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE</td>
<td>Create</td>
<td>Model</td>
<td>File</td>
</tr>
<tr>
<td>TEXT</td>
<td>Create</td>
<td>Template snippet</td>
<td>Textblock</td>
</tr>
<tr>
<td>FOREACH</td>
<td>Create</td>
<td>Model</td>
<td>Model</td>
</tr>
<tr>
<td>FOREACH</td>
<td>Create</td>
<td>Model</td>
<td>Textblock</td>
</tr>
<tr>
<td>EXPRESSION</td>
<td>Create</td>
<td>Model</td>
<td>Textblock</td>
</tr>
</tbody>
</table>
```

Figure 6: Derivation of traceability data out of Xpand statements

Having explained the substrate mapping for the pointcut definition, we turn to the associated advices, which integrate the oAW framework into the GTI by creating traceability data and populating this data to the traceability repository. Figure 1 sketches the oAW integration exemplary for the $FILE$ statement.

```
@Aspect
class OawConnector extends AbstractTransEngineConnector {
    @After("execution(* * evaluate( .. ) && this(fs))
        && args(ctx)")
    public void executeModelOperationInternal (FileStatement fs, XpandExecutionContext ctx) {
        Model Artefact Artefact sourceArtefact = new Artefact(...) ;
        // File Artefact Artefact targetArtefact = new Artefact(...) ;
        // Trace Link TraceLink link = new TraceLink(sourceArtefact, targetArtefact, CreateTraceLink);
        this.executeModelOperationInternal(link);
    }
}
```

Listing 1: Pointcut definition for FILE statement

Recalling from Section 4.2, the connector’s responsibility is the invocation of the $executeModelOperationInternal()$ method in order to populate traceability data. Accordingly, the $executeModelOperationInternal()$ method (line 6) is implemented by weaving it after the execution of the $evaluate()$ method of the FileStatement. This triggers the creation of source (of type Model) and target (of type File) artefacts based on the oAW framework-internal XpandExecutionContext (line 8). Finally, a CreateTraceLink between both artefacts is established by calling the $executeModelOperation()$ method of the GTI.

5.2 QVT connector

For the QVT connector, we used Operational QVT, which offers a dedicated traceability support by generating a so-called trace file for every model transformation. While executing the transformation, operational mappings are logged into the trace file. The format is defined by the QVT internally used traceability metamodel.
Following the line of argumentation from Section 4 to use the advantages of integrated traceability solutions, we make use of the existing traceability data contained in the QVT trace file. Our approach to extract traceability data occurs in two steps: First, a model transformation (see listing 2) is executed, allowing the transformation of any instance of the QVT traceability metamodel to an instance of the TraceDSL by defining a mapping between both metamodels. Second, we import the information of the TraceDSL instance to the desired traceability repositories.

Listing 2: Trace File to Trace-DSL Instance Transformation

```java
main() {
  var facetsList := List {
    self.map TraceRecord2TraceLinkNameFacet(),
    self.map TraceRecord2TraceLinkMappingFacet()
  }

  sources := self.map TraceRecord2SourceArtefact() -> asList();
  targets := self._result._result.value.modelElement.map EObject2UUID();
  facets := facetsList;

  mapping TraceRecord::TraceRecord2SourceArtefact() {
    artefact {
      uuid := self._result._result.value
      modelElement.map EObject2UUID();
      facets := self
      .map TraceRecord2SourceArtefactTypeFacet() -> asList();
    }

    mapping TraceRecord::TraceRecord2TargetArtefact() {
      artefact {
        uuid := self._result._result.value.
        modelElement
        .map EObject2UUID();
        facets := self
        .map TraceRecord2TargetArtefactTypeFacet() -> asList();
      }
    }

    mapping TraceRecord {
      CreateTraceLink {
        init {
          var facetsList := List {
            self.map TraceRecord2TraceLinkNameFacet(),
            self.map TraceRecord2TraceLinkMappingFacet()
          }

          sources := self.map TraceRecord2SourceArtefact() -> asList();
          targets := self._result._result.value
          .map EValue2TargetArtefact(self) -> asList();
          facets := facetsList;

          mapping TraceRecord::TraceRecord2SourceArtefact() {
            artefact {
              uuid := self._result._result.value
              .map EObject2UUID();
              facets := self
              .map TraceRecord2SourceArtefactTypeFacet() -> asList();
            }
          }
        }
      }
    }
  }
}
```

6. FUTURE WORK

The future work of our research is directed at continuing the development of our generic traceability framework for augmenting arbitrary transformation engines with a traceability mechanism based on the TraceDSL. In accordance herewith, we envision our framework to consist of three main components: a) a traceability metamodel; b) an extensible trace manager, that handles the part of serialisation and management (import, export, visualisation, analysis, edition) and c) a language agnostic TraceDSL.

Regarding the TraceDSL, the following points are to be investigated with respect to defining rules for controlling traceability data: a) Artefact- and Link types: Rules on when to trace which kind of artefact types from a defined type system by using which kind of link types from a defined type system; b) Granularity and Scope (cf. Section 2): Rules on the link granularity, as well as, on the scope for which traceability data is recorded, are derived with respect to the aforementioned type systems; c) Link Cardinality: Rules on the cardinality of link types; d) Traceability Time: Rules to configure the period of time for which a given artefact is to be traced.

The focus of this investigation is to use these rules to improve the formulation of traceability queries with regard to a certain traceability scenario. Thereby, we assume a traceability query to be dependent on the semantics of the traceability data. For instance, while filtering the data for certain java packages the corresponding artefact type needs to be known for the query formulation. Therefore, we wish to abstract as much as possible from artefact and link types bound to their native reference and rather define more generic types, e.g. to introduce a generic artefact type ClassArtefact as opposed to the types C++Class as well as JavaClass for corresponding artefacts. Consequently, specific types would be negligible with respect to query formulation and thus more user-friendly, provided that the generic types account for the successful execution of traceability queries.

Secondly, regarding the usage of facets, it is under exploration to derive a facet-based classification, based on the identified artefact and link types from above, given that facets are orthogonal dimensions of a model and all classes in facets are independent, i.e. do not know of each other.

Further work aims at augmenting arbitrary model transformation approaches with a traceability mechanism. Different augmentation mechanisms are to be explored, classified and aggregated. This initiative is part of the requirements analysis as a development directive for model transformation approaches which are bound to incorporate a traceability mechanism.

7. CONCLUSION
In this paper we proposed a generic traceability framework for augmenting arbitrary transformation engines with a traceability mechanism based on a language agnostic Trace-DSL. Essentially, our framework is based on a generic traceability interface, which provides the connection point for arbitrary transformation engines and traceability engines.

To demonstrate the feasibility of our framework we have shown two augmentation mechanisms resp. developed trace connectors, thereby choosing a representative from the explicit (i.e. oAW) as well as implicit (i.e. QVT) trace link generation class. The augmentation mechanism of the generator logic of oAW is based on aspect-oriented programming in order to automate traceability data extraction. Consequently, the minimization of efforts to achieve traceability is reached as per problem definition (cf. Section 3). After taking into account the initial effort of developing the oAW connector, the following advantages hold: a) no likely error-prone and time consuming factors of manual encoding of traceability-specific rules, b) no recurring efforts from a) for individual model transformations c) no pollution of transformation programs. The second augmentation mechanism is based on a model transformation for QVT with the advantage of making use of the already integrated QVT traceability solution, while still guaranteeing traceability data conforming to the Trace-DSL, which is needed to account for the fine-tuning of traceability data mandated by the traceability scenarios.

Furthermore, our approach is based on a domain-specific language for traceability, to ensure a unified traceability metamodel (cf. problem definition, Section 3) and with the maxim to account for a traceability metamodel extensibility mechanism based on facets. The latter is to achieve an adequate traceability data expressiveness to hold for the traceability scenarios in Section 2.

8. REFERENCES

http://www.eclipse.org/m2m/atl/.
[12] openArchitectureWare.
http://www.openarchitectureware.org/.
[18] Xpand.
http://www.eclipse.org/modeling/m2t/?project=xpand.
Toolling for the Full Traceability of Non-Functional Requirements within Model-Driven Development

Anton Yrjönen
VTT Technical Research Centre of Finland
Kaitoväylä 1, PL 1100
90570 Oulu
+358 40 821 0493
Anton.Yrjonen@vtt.fi

Janne Merilinna
VTT Technical Research Centre of Finland
Vuorimiehen tie 3, PL 1000
02044 VTT
+358 40 074 0058
Janne.Merilinna@vtt.fi

ABSTRACT
There is an ever-increasing need to rapidly deliver products, whilst, at the same time, also delivering products of high quality. To improve the quality of products and increase productivity within software development processes, all phases of the development process must fit together well. While defining requirements for the system, it must be ensured that the correct requirements are defined as well as ensure that they can be translated into a design fulfilling the requirements. The earlier the correct requirements are found, the easier and cheaper it will be to design good products. Finally, the design must be verified against the correct requirements. To realize this, requirements traceability is of extreme importance for development processes. The non-functional requirements (NFR) are particularly important and difficult. In this paper, we will report on an integrated tooling solution for a Domain-Specific Modelling approach that enables and guides towards defining accurate and non-conflicting requirements. Additionally, the solution enables a full bi-directional traceability from the requirements to models to the implementation, and offers an up-to-date overall view of the state of the requirements within the product.

Categories and Subject Descriptors

General Terms
Design, Languages, Verification.

Keywords
Domain-Specific Modelling

1. INTRODUCTION
In software processes and tools, there is often a semantic and practical gap between requirement engineering (RE) and software engineering (SE), due to the very different language and abstraction. Requirements are typically presented in a natural language, using problem domain concepts, whilst software design merely uses programming languages and concepts. The non-functional requirements (NFR), which may not easily and directly map into specifications, are particularly important and difficult. In addition, they can be system-wide aspects. Bridging the gap is an essential condition for increasing productivity and product quality in the software (SW) industry. SW processes must be capable of creating proper requirements, crafting software that matches these requirements and validating it. This requires a seamless collaboration of those activities concerning requirements analysis, validation and management, as well as software production.

Requirements traceability refers to the ability to describe and follow the life of a requirement, in both the forwards and backwards direction (i.e. from its origins, through its development and specification, to its subsequent deployment and use, and through all the periods of on-going refinement and iteration in any of these phases.) [1] Based on the previous definition by Gotel and Finkenstein, the traceability appears to be a formulation of the problem about the gap. Instead of explicitly crafting separate traceability tools, techniques and methods, there is also an option to create such design tools that include all the necessary traceability capabilities themselves.

Model-Driven Development (MDD) and especially Domain-Specific Modelling (DSM) seem to be prominent technologies to achieve a higher productivity and fewer errors in software production. The increase in productivity and decrease in programming errors is achieved by shifting the abstraction level from the solution-space to the problem-space, thereby enabling the modeller to work with elements and rules of problem-space instead of classes, methods, variables, and rules of programming languages. In DSM, it is common to take advantage of a complete code generation, in the sense that it is not needed, nor should be, to modify the generated code. The DSM approach in industrial cases has shown an increase of a factor of 5-10 in productivity [2]. Although the witnessed gains seem to be impressive, the linkage between RE and design still needs to be added, in order for the full software development processes to become as efficient as possible. Due to the decision to use problem domain concepts instead of e.g. programming language concepts in the application design, DSM itself seems to be a tempting possibility for closing the gap and improving traceability.

According to literature [1][3][4], requirements traceability can be classified to pre-RS and post-RS traceability, based on their appearance in relation to the requirements specification (RS). In addition, for both classes, there can be separate sub-classifications by the direction in relation to the requirements specification; backward-from and forward-to traceability relates to traceability between business domain and requirements documents, thus belonging to the pre-RS scope, whereas forward-from and backward-to are positioned to the post-RS division.

In the pre-RS and post-RS division, requirements derived from other requirements are positioned to the pre-RS scope. Typically existing requirements management tools can handle the pre-RS scope well, and many also try to provide the post-RS
traceability. However, the post-RS traceability usually crosses the tool and semantic borders in such a manner that provides no backward-to traceability, due to the fact that design components are crafted in separate tool environments using separate languages and paradigms that do not offer necessary linkage. The normal practice to solve this issue has been to create separate design documents that are stored in requirements management (RM) databases (RMDB). However, the distinction of documents and actual software implementation is vulnerable to losing synchronization without excessive and laborious documentation maintenance work throughout the whole process. This, on the other hand, breaks the traceability. In order to avoid losing the trace link from requirements to the rest of the development phases, requirements must be linked directly to the output artefacts of the rest of the phases.

In addition to direct interdependencies in the time axis (Pre-RS/Post-RS), there is additional complexity amongst NFRs that are of system-wide aspects, such as resource consumption or performance. These interdependencies should be traceable as well. e.g. one should find any other design model that influences other NFRs than the one under examination, should there be a problem or conflict in realizing some requirements.

The NFR Framework [5] by Chung et al. is a set of goal-based methods and tools for RE, that tackles the difficult NFRs by decomposing them into sets of smaller subgoals, revealing their interdependencies and helping to refine them into concrete specifications (operationalizations). The NFR Framework is a good tool for RE and managing the NFRs, but does not solve the full complexity of the traceability problem as it is not integrated into the rest of the development by tools or data. Softgoal Interdependency Graphs (SIG) are a central part of the NFR Framework, containing the decomposed softgoals (NFRs) and their operationalizations. There are also catalogues that store knowledge on the typical interdependencies and operationalizations of the NFRs.

In [6], an implementation of the NFR Framework in MetaCase, MetaEdit+ language workbench was presented, including an automatic evaluation for analyzing SIGs. It also extended the NFR Framework with a concept of quantifiable 1 NFRs to form a NFR+ Framework. The quantifiable NFR provides evidence-based information for the use of a requirements engineer to determine whether the defined NFRs are achieved or not. They also serve as a connection point and guidance to software designers, by stating the desired outcome. It was argued that the quantifiable NFRs introduced are one important step to close the gap between RE and SE. In this paper, we will take a look at how such a framework supported by tools can enable the full traceability of NFRs in the context of DSM.

This paper is structured as follows. In Section 2, the technical approach and implementation are explained thoroughly. In Section 3, the usage of the framework and accompanying tools are demonstrated with a case example easy to comprehend. Finally in Sections 4 and 5, the results are discussed and the paper concluded.

2. THE NFR+ FRAMEWORK WITH FULL TRACEABILITY

In this paper, the full traceability is considered as the forward and backward traceability of the whole SW development process. The tooling to be introduced in this section is designed to enable such traceability.

---

1 We use the term quantifiable instead of measurable, as sometimes the evaluation of a requirement is enough.

2 http://sourceforge.net/projects/osrmt/
2.2.2 Exporting the Requirements to the OSRMT

It is also possible to change the requirements within ME+ (for example, update descriptions or any other data fields) and export them from ME+ to OSRMT. When an export is executed for a requirements graph, the export generator produces an XML file in the OSRMT format, containing each of the requirement objects within the graph. The name and path for the XML file is queried from the user. If a requirement contains a SIG or multiple ones, they are traversed during the export procedure and all the related operationalizations and quantifiable NFRs are collected into the description field of the requirement. This auto-generated decomposition information is marked with double square brackets $[[[\text{ ]}]]$. During the import to OSRMT, these brackets are used to separate auto-generated information from a manually created description. If the requirements are imported back to the NFR+ Framework, the brackets are removed from the requirement description to avoid importing out-of-date decomposition declarations.

The requirements are elaborated via an explosion link from the original requirements to SIGs. The requirements can be elaborated in the same way as presented in [5], as the SIG of the NFR+ Framework provides the same facilities as the NFR Framework.

2.2.3 Requirements Elaboration

In addition to the same facilities as the original NFR framework, the NFR+ Framework also makes it possible to incorporate so-called quantifiable requirements. The quantifiable requirements can be connected to a softgoal in SIG, as presented in Figure 3. The meterization symbol showing a yellow segment in the middle of Figure 3 presents an undefined result of a quantifiable requirement connected to the softgoal of a type throughput. The $=$ symbol in the object indicates the contribution of the measurement to the parent softgoal. In Figure 3, it is equal to but can be changed to any of the normal softgoal contribution signs.

2.2.4 Catalogues and Tools

Different catalogues, e.g. contribution and method catalogues [5], are part of the NFR Framework. The catalogues in the NFR+ Framework are implemented as individual model types. In addition, tools which automatically browse these catalogues and assist the user in different tasks are implemented. The NFR Framework [5] includes type catalogues, correlation catalogues and method catalogues. We have also added a contribution catalogue to the NFR+ Framework.

The type catalogues contain stored knowledge on the possible type decompositions for softgoals. Each softgoal object has a property type. These properties are actually instances of a class NFR Type. NFR Types can be created and stored in Type Catalogue graphs, wherein they can be constructed as a tree structure.

The method catalogue lists the possible operationalizations for softgoals. The catalogue is utilized to store knowledge about possible solutions to satisfice the softgoals.

The correlation catalogues contain information on the implicit interdependencies amongst softgoals, for example, such that encrypting data tends to slow down communication. The same can be expressed using the NFR Framework terminology, e.g. a softgoal of type encryption poses a HURT contribution to the softgoal of type communication speed.

2.2.4.1 Contribution Catalogue

The contribution catalogue is a special SIG-alike graph that defines the outcome of pairs of child labels and contributions. The SIG evaluation generator attempts to find this special graph from the currently active project in the ME+ environment and checks the resulting parent label from the diagram if such is defined. Otherwise, defaults are used according to the NFR Framework [5]. The contribution catalogue can be modified to influence the evaluation. Figure 4 illustrates the contribution catalogue. To read this catalogue, one first has to look for the desirably labelled child (lower) softgoal from amongst the pairs, then find the correct contribution symbol, and thus the parent (upper) softgoal label shows how such a combination is to be evaluated. By modifying the contribution catalogue, the user can affect how the softgoals are labelled during the automatic evaluation of a SIG.

Figure 3. Attaching a quantifiable requirement to a softgoal.

The three different meterization symbols are fail (red), pass (green) and undefined (yellow). The undefined state exists if the connected quantifiable NFR has not yet been evaluated within any of the design models.
Implicit Interdependency Assistant

Correlation catalogues (see Figure 5) are utilized for storing knowledge on the contributions of typical implicit interdependencies between different softgoal types and/or operationalizations. It is best represented as a matrix, where each column and row is occupied with different softgoals/operationalizations, and there is a corresponding contribution sign in their intersection. When desired, an assisting tool can be executed that browses through the catalogues and locates defined correlations for softgoals/operationalizations. These are then listed for the user to consider if they are to be acknowledged within the SIG or not. For the sake of simplicity, they are not automatically added to the diagrams but only shown as propositions for the user to fill in if appropriate.

Decomposition Assistant

The Decomposition Assistant tool, available in SIGs, can be utilized for searching all type and method catalogues for possible type and operationalization decompositions for selected softgoals and operationalizations. The Decomposition Assistant lists all the possible decompositions which are not yet implemented in the SIG. The user can then add proposed decompositions if they are considered appropriate.

SIG Analysis Tool

The SIG graphs can be evaluated upon a user decision through a specific generator which traverses the model and generates a model transformation script in Python. The generated transformation script calls the SOAP API of ME+ in order to execute commands related to changing the labels of the softgoals appearing in SIGs. This arrangement is required, since it is currently not possible to alter property values directly from a code generator.

Application Design View

The operationalizations and quantifiable NFRs defined in the SIGs will be attached to the separate application models to pinpoint the exact parts of the software that they relate to, or are implemented in. An example of how this is done is shown in Section 3.

NFR Traceability in the NFR+ Framework

Traceability Link Creation

The SIGs are a central mediator of the traceability from early requirements to specifications and an implementation in terms of application models. The activity of constructing the SIGs is virtually the activity of constructing the traceability links. Another kind of explicitly created traceability links are those graph explosions that associate certain SIGs with requirement objects. In addition, implicit traceability exists through the object references within the ME+ internally. It is worth noticing, that all this simultaneous traceability is created automatically, while just creating the requirements and elaborating them towards specifications and restrictions, and while designing and modifying the applications according to these requirements and specifications. No separate, additional traceability linkage tools are required. All of the models are intertwined, so that the traceability is inherently present throughout all the activities.

Pre-RS

Part of the Pre-RS traceability is stored in SIG graphs that describe and document the history of eliciting the final requirements in terms of operationalizations and quantifiable NFRs. This is bi-directional, i.e. requirements can be traced back from operationalizations to early stakeholder requirements, and vice versa. Part of the Pre-RS traceability crosses the tool boundary in cases where there are imported requirements originated from another tool. In such cases, there is a possibility to provide forward traceability from external tools to the NFR+ Framework through e.g. hypertext links to...
documents stored elsewhere and created by the NFR+ Framework tool. In such a case, backward traceability would require manual browsing from the RMDB or some additional, specific integration steps that do not exist in the current implementation. However, all relevant information can be imported and maintained in the NFR+ Framework for full traceability.

2.4.3 Post-RS
Post-RS traceability is implemented through the connections between SIG and design models, i.e. namely operationalizations, and quantifiable NFRs. Traceability is automatically up-to-date all the time, and navigable to both directions. In addition to simply manually navigating through the object bindings in the graphs, there is also some visible information available about the traceability. In the application design view, operationalizations report on the impact to the NFRs as observable in Figure 12 (see the text under thick-clouds, i.e. operationalizations). In SIGs, the meterization symbols report the verification results (see Figure 13, meterization symbols in particular).

2.4.4 NFR Cross-connections
The NFR interdependencies are maintained internally within NFR+ Framework projects in ME+ repositories. This may be a complex network of interdependencies, not only through SIGs but also via connections to design models. These cross-connections provide a traceability that is neither backward nor forward in type. For example, the same application model may involve several various NFRs defined in separate SIGs. Applying some design decision proposed by one SIG, might simultaneously affect the evaluation of another SIG through a changed system behaviour or structure. This kind of interdependency can also be traced via the interconnections provided with backward/forward traceability. One could describe these for example as forward-backward or backward-forward traceability.

3. CASE EXAMPLE
To demonstrate the NFR+ Framework, the bread baking manufacturing process is applied as a demonstrator. The utilized demonstrator is only for demonstration purposes and it does not reflect any real baking factories. An example of such a baking process is depicted in Figure 7.

![Baking process](image)

**Figure 7. Baking process.**

Figure 7 can be read as follows. First, 50kg of raw material for whole meal bread is produced. Second, the raw material is delivered to the “main rolling station” where the bread is to be rolled. The raw material flows to the main rolling station at 1kg/5s. In the rolling station, the bread is baked. After that, the buns are delivered to the main oven. The oven bakes 50kg of bread at 220°C for 10min. The bread is then delivered to the packaging department where 10kg of buns are packaged in 10min.

3.1 Requirements Elaboration
For the baking process, the following requirement is defined in the starting phase and imported to the ME+ from OSRMT:

- Bake good wholemeal bread rapidly and cost-efficiently.

Three abstract softgoals can be identified: “good bread”, “bake the bread rapidly” and “cost”. The rapid baking process softgoal can be decomposed into a SIG depicted in Figure 8. Rapid baking decomposes into the high throughput of an oven, high throughput the baking, i.e. producing raw material rapidly, and the high throughput of rolling. The high throughput of an oven is further decomposed into a rapid baking and high volume oven. Using a high temperature might hurt the “good bread” softgoal and hurt “costs” as well.
In Figure 9, quantifiable requirements and constraining operationalizations are added into the SIG. For a high throughput of baking, i.e. producing the raw material, a “1kg of raw material should be produced in under 60s” requirement is defined. For rapid rolling, “1kg of raw material should be handled in under 60s”. For a high volume oven, a minimum volume requirement of 50kg is defined. For “rapid baking” a hard requirement is defined that states that 10min is the maximum time to bake bread in an oven. Considering these, for rapid baking in an oven of a quantifiable throughput requirement of 50kg/10min is defined. For a high temperature oven, 220C is defined, as this is the maximum temperature for baking wholemeal bread in this example. As this can also be considered as a restriction to temperature, an operationalization for temperature is stated. This operationalization can be considered to hurt satisficing the high temperature softgoal, as it is considered that 220C is not high enough.

Every leaf softgoal has their quantifiable requirements (considering bake rapidly softgoal). Therefore, the original requirement, i.e. “Bake good wholemeal bread rapidly and cost-efficiently” can now be refined to: “Bake good bread rapidly by: 1) Producing 1kg of wholemeal bread raw material in 60s, 2) Rolling 1kg of bread in 60s, and using a temperature of no more than 220C temperature in the oven where the volume of the oven should be at least 50kg and the buns should be baked for no more than 10min and in this way, the throughput of an oven should be 50kg of bun in 10min, cost-efficiently”. This can be considered to be a requirement where there is a clear pass/fail criterion and there should be no more ambiguities. To be noticed, “good” and “cost-efficiently” should be elaborated upon, but for the sake of clarity, they are omitted in this demonstration.

After assigning the operationalization to the SIG, the SIG needs to be evaluated in order to know whether the current solution enables to satisfice the requirements of the baking process. As the NFR+ Framework tool provides an automated evaluation for SIGs, only a push of a button is needed to automatically evaluate the SIG. The evaluated SIG is presented in Figure 10.

After the evaluation, it seems that the “Bake bread rapidly” softgoal is in a conflicting state. The reason for this conflict is caused by a conflict between using the convection oven and only heating at a temperature of 220C. The reasoning for this conflict is that applying the convection oven does have a positive impact on rapid baking, but at the same time, 220C might not be enough. The only way to resolve this conflict is to test whether 220C is enough when a convection oven is applied. In this situation, we do know (browsing the catalogues) based on previous experience that this is enough, thus we can override the conflicting “Rapid baking” softgoal with a “satisfied” value. The SIG can now be re-evaluated. The re-evaluated SIG is satisfactory and ready to be implemented, as shown in Figure 11.
3.3 Design and Implementation

Considering the requirements produced in the previous phases, an application can be modelled. Such a model is depicted in Figure 7. The utilized operationalizations are added to the model. The modeller looks for these operationalizations from the corresponding SIG. The operationalizations also automatically inform what their impact on the neighbouring softgoals is (see Figure 12). This enables the modeller to not only see that she is using operationalization defined in the requirements engineering phase, but also why those operationalizations are utilized.

3.4 Testing

Next, the modelled system should be tested. Thus, the placeholders for measured (or evaluated) values are attached to the model such as depicted in Figure 12. The quantifiable requirements also express the current status of the realization of the requirements. When there is a red tag on the right side of a requirement, the requirement is not satisfied. In cases where the requirement is satisfied, such a tag disappears.

After the placeholders are attached to the model, an implementation can be developed. Usually in the case of DSM, the implementation is generated from a model. Values for the place holders, a.k.a. measurement mechanisms, can be set manually or automatically, such as done in [7]. In the case of this baking process, naturally no such process is actually developed and the measurements are only for demonstration purposes. Figure 12 depicts the baking process where measured values are “measured” from the running baking process.

![Figure 12. The baking process with measured values.](image)

In Figure 12, quantitative requirements for a rapid rolling and the high throughput of an oven are satisfied (there are no red tags in the requirements). However, rapid baking seems to be unsatisfied. The status of the quantitative requirements is now also automatically propagated to the SIG as depicted in Figure 13(see the meter pointing to the red).

To evaluate the impact of the current status of the quantitative requirements, the SIG can be automatically evaluated as discussed above. By pressing the evaluation button, the SIG updates as presented in Figure 13. It is now easily observable that although two requirements were fit, the third requirement concerning baking causes the original requirements to not be satisfied. This result clearly indicated that there is no other way to overcome this (see interdependencies) except by improving the throughput of the baking.

![Figure 13. Evaluated SIG after testing.](image)

Considering the traceability, the connected operationalizations and quantifiable requirements provide full backward and forward traceability. They relate individual the softgoals and branches of a SIG tree into specific parts of the application model. Thus, one can select any operationalization and follow the links back to the original stakeholder requirements. Or vice versa, starting from any of SIG’s topmost softgoals, one can look for references to related operationalizations and find all the parts of implemented design model(s) which these stakeholder requirements are affecting. The same also applies for the quantifiable measurements.

4. DISCUSSION

The presented approach provides the traceability of NFRs from early user requirements to implementation and testing, and backwards. We call this full traceability. The NFR Framework itself creates traceability from early NFRs to specifications such as operationalizations. Additional quantifiable NFRs defined in the NFR+ Framework also provide a transparent traceability between test specifications and design targets for NFR softgoals.

By using the preset-end approach, the managing of system-wide aspects becomes easier, since the related NFRs are all connected to every relevant design model parts. Thus, it is possible to find every design detail that shares the same NFR types or that have an impact on the same softgoals. Using the ME+ for implementing the NFR Framework with added extensions seems to be worthwhile. The ME+ tool has some limitations, with the worst of them being the inability to modify the models through the code generators. We were however able to circumvent these limitations by exploiting other features of the tool, i.e. ME+ SOAP API. This was a bit tricky and laborious, but do-able. It was easy and straightforward to create the SIG and catalogue meta-models to implement the NFR+ Framework. The ability to seamlessly combine the domain-specific application models and the NFR+ Framework models is a positive feature of the well-supported metamodelling facilities of ME+. This helped us in implementing the traceability, since the objects and their relationships are already traceable by the ME+ tools. It was enough to include the quantifiable requirements and operationalizations into the design graphs to achieve full traceability.

Compared to our previous work [6], we have included the import/export from external RMDB which makes the presented
approach more compliant with existing product development
designs. Another significant increment is the addition of the
operationalizations into the application models with the notion
of affected softgoals. This provides a backward traceability that
was not directly available in earlier versions.

For future work, there still remains interesting possibilities.
Getting into the details of the executable code in run-time
tracing would benefit in e.g. performance tuning. The
scalability of the approach should be further studied with non-
trivial applications and domains. There is a possibility that
overly complex NFR interdependencies would clutter the
design graphs if everything is shown at the same time.

5. CONCLUSION
In this paper, a solution was presented for managing traceability
in a product development. By adapting the NFR+ Framework,
DSM application development and integration with
requirements management tools full, bi-directional, NFR
traceability from early requirements to design and testing was
introduced. The solution is based on an integrated modelling
approach where the modelling techniques are applied from the
beginning of the process, and carried on completely within an
integrated development environment.

The history of requirements and specifications is available
through the SIGs. SIGs also represent an overview to the status
of NFRs in every phase according to the best information
available; in early phases, either designers’ argumentation or
general knowledge stored in catalogues. In later phases, the
measured or evaluated empirical data was also specifically for
the implemented system.

From design models, one can trace the full requirement history
through operationalizations and quantifiable requirements.
From SIGs, on the other hand, one can locate every design
entity that is relevant for the given softgoal/requirement.

The NFR+ Framework was demonstrated with a simple case
example, describing the bread baking processes and
requirements for it. The example shows how ambiguous NFRs
can be transformed into detailed specifications and testing
criteria.

6. REFERENCES

Requirements Traceability Problem, Proceedings of 1st
International Conf. on Requirements Engineering, IEEE,

– Enabling full code generation, John Wiley & Sons, New

requirements traceability in an industrial environment,
Fifth IEEE Symposium on Requirement Engineering,


Requirements in Software Engineering, Springer, Reading,
Massachusetts, 2000.

Framework with Measurable Non-Functional Requirements.
2nd International Workshop on Non-
functional System Properties in Domain Specific Modeling

Quality-Driven Domain-Specific Modelling, Proceedings
of the 9th OOPSLA Workshop on Domain-Specific
Modeling (DSM09)

[8] Ebert, C., Putting requirement management into praxis:
dealing with nonfunctional requirements, Information &
A State-Based Approach to Traceability Maintenance

Nikolaos Drivalos-Matragkas
Department of Computer Science, University of York
York, UK
nikos@cs.york.ac.uk

Dimitrios S. Kolovos
Department of Computer Science, University of York
York, UK
dkolovos@cs.york.ac.uk

Richard F. Paige
Department of Computer Science, University of York
York, UK
paige@cs.york.ac.uk

Kiran J. Fernandes
The York Management School, University of York
York, UK
kf501@york.ac.uk

ABSTRACT

Traceability of software artefacts has been recognized as an important factor for supporting various software development activities. However, establishing traceability requires a substantial investment in effort. Even when an initial set of traceability links has been established, this set is subject to gradual degradation as the associated artefacts are modified, e.g., due to the evolutionary nature of software development. To avoid this, traceability must be constantly maintained and evolved. The manual maintenance of traceability can be time consuming and error-prone. This paper focuses on reducing the manual effort incurred in performing traceability maintenance tasks. This is achieved by introducing a dedicated mechanism in the Traceability Metamodelling Language, which is used for detecting and evolving problematic trace links. A concrete example is used to demonstrate the practicality and usefulness of our approach.

Categories and Subject Descriptors
D.2 [Software]: Software Engineering ; D.2.5 [Software Engineering]: Testing and Debugging—Tracing

General Terms
Management

Keywords
Model Driven Engineering, Traceability, Evolution

1. INTRODUCTION

The second law of thermodynamics states that a closed system’s disorder cannot be reduced, it can only remain un-
changed or increase. A measure of this disorder is entropy. This law also seems plausible for software and its associated artefacts, such as documentation or design. As software is modified, its disorder always increases. This is known as software entropy [3]. With the advent of iterative and agile methods of software development, software entropy becomes more of an issue, since change is more frequent. If the relationships between all artefacts are documented and maintained, the management of change and evolution can potentially be made more efficient. Any relationship that exists between the artefacts involved in the software development life cycle is called traceability [2].

Since traceability information is tightly coupled with the other artefacts of the development process, it is subject to software entropy as well. That is, traceability information is subject to gradual degradation as the related artefacts are modified. As a result, the recorded relationships may end up being incorrect or inaccurate and as a result cannot support change propagation or impact analysis activities. Manual maintenance of traceability can be time-consuming and error-prone due to the large number of potential relationships that exist even for small software systems. In this paper we present an approach to semi-automatic maintenance of traceability information. The proposed approach is part of a holistic approach to traceability, the Traceability Metamodelling Language (TML) [9].

The rest of the paper is organised as follows. In Section 2, we discuss background material related to the work presented in this paper, while in section 3 we identify a set of requirements for a traceability maintenance approach. In section 4 we propose the main contribution of this paper, which is a state-based approach to semi-automatic traceability maintenance, while in Section 5, we present a concrete example that demonstrates the practicality and usefulness of our approach. Finally, in section 6 we provide a discussion on related work and in Section 7 we conclude and identify interesting further work on the subject.

2. BACKGROUND

One of the most challenging aspects of traceability is how to maintain the integrity of the relationships, i.e. trace links, while the referenced entities continue to change and evolve. To achieve this, referential integrity and link integrity must
not be violated. The difference between referential and link integrity is very subtle. In the context of relationship management, referential integrity is a measure of the reliability of a reference to an end-point, whether a source or a destination of the relationship. On the other hand, link integrity measures the reliability of the whole link (i.e., all endpoints) [8]. When the entity at the end of a link is not present or is not the entity that was intended by the link author, then the referential integrity is violated and the link is said to be dangling. Both manual effort and computation time need to be invested to ensure referential integrity and the goal is to minimize manual effort at the expense of computational time. Ensuring referential and link integrity requires the examination of both the links and the changes of the models under consideration. In sections 2.1 and 2.2, we analyze these two parameters of the traceability maintenance issue.

2.1 Link Types

[2] argues that the way a relationship is managed depends on the type of the relationship. More precisely, [2] identifies two types of relationships or trace links. An imposed relationship is a relationship between entities that exists by volition of the relationship creator and it can not be invalid until its creator determines that it is invalid. For example a satisfies relationship can be created between a requirement and an element of a design model. When the requirement changes it is not clear if the relationship is still valid.

On the other hand, an inferred relationship is one that exists because the related entities satisfy a rule that describes the relationship. An inferred relationship cannot be invalid since if the rule is not satisfied the relationship does not exist. An example of an inferred relationship is the relationship which is a result of a model transformation.

2.2 Model Change Types

Knowing just the type of a link is insufficient for all traceability maintenance scenarios. To maintain a traceability relationship efficiently, one also needs to know the types of model changes that take place at the trace link ends. For instance, in the case of an imposed relationship, a link can be invalidated without input from its creator, if one of the entities, which are referenced by the link ends, is moved or deleted.

There are three possible model change types — addition, modification or deletion. These model change types are similar to the create, read, update and delete (CRUD) functions of persistent storage. In the literature, many additional types of change have been identified, such as model element replacement or model merging (for example in [18]). In the context of this work, we consider all other types of change as a combination of the three aforementioned atomic types. For example, a model element replacement consists of two different actions, element deletion and element addition. Similarly, model merging can be considered as the compound effect of deleting two models and adding a new one, which is the union of the two.

Addition of models or model elements is not a source of dangling links. However, it affects the completeness of the link set. Namely, new models or model elements may be added and all the appropriate links to and from the new entities are not added. Deletion can result to link integrity errors since the link end does not point to an entity. Finally, modification can occur at two different levels. First, whole models can be modified. This includes the moving of the model from one directory to another or the renaming of the model. Second, internal modifications can take place at the model element level. Such modifications include the change of element attributes in a way that the validity of the link is compromised. For example, in a model transformation scenario, if we change the type of an element in the source model, then the trace link with the relative element in the target model might not be valid any more.

2.3 Classes of Solutions

There are two main approaches for maintaining trace link integrity: event-driven and state-based approaches. In the former approach, the elementary changes of the various model elements are constantly monitored, change events are generated based upon these elementary changes and finally according to the change events corrective actions are taken. In the latter approach, the detection of model changes takes place by comparing different versions of the models and suspect links are found based on the identified changes.

2.3.1 Event-Driven Traceability Maintenance

This approach revolves around the monitoring of elementary model change events and the generation of compound changes. This is achieved by utilizing a set of rules for recognizing the events as constituent parts of intentional development activities. Once these activities have been identified, traceability links related to the changing model elements can be updated automatically.

The atomic model changes associated with the first phase of this approach are usually the ones mentioned in Section 2.2, namely addition, deletion and modification. In the second phase, the development activity that is realized by a chain of the aforesaid changes must be recognized. Such activities include replacement, merging and splitting of model elements. Atomic changes are associated to development activities using a predefined set of rules.

There are three main challenges associated with recognizing the various compound development activities [17]. First, the same development activity can be achieved by different elementary changes. For example there are two different ways to replace a model element. The developer can either delete the original element and then add the new one or she can just modify the existing one. An additional challenge is the fact that the same development activity can be achieved by the same elementary changes in different sequences. If for example a developer wants to replace a model element, she can either add the new one first and then delete the old one or vice-versa. Finally, the type of change and the impacted model element do not offer enough information for relating changes to each other. The final step of the event-driven approaches to traceability maintenance is the reconciliation of the dangling links. This can be performed manually by the user or automatically if a predefined rule exists for every change event.

To overcome the identified challenges, the event-driven approaches restrict their scope in two main ways. First, the user is restricted to using a particular tool for evolving the models. Second, these approaches restrict their scope to a particular notation. Undeniably, if those methods are used in the domain for which they were build, they can be efficient in preserving trace link integrity. Nevertheless, event-driven approaches are not suitable to provide a generic
solution to traceability maintenance in the open-ended and ever-changing world of Domain Specific Languages (DSLs) due to the aforementioned restrictions. For example, in [5] traceability information between requirements is maintained. One limitation of this work is the fact that the requirements have to be evolved using a particular notation so that the tool is able to identify the various change events. Although, the approach proposed by [5] can be efficient for maintaining traceability between requirements, it cannot be generalised to other artefacts or DSLs since the tool and the notation are suitable only for structured natural language (in this case requirements).

2.3.2 State-Based Traceability Maintenance

In this method, the detection of model changes takes place by comparing an instance of the model under consideration in time \( t \) with an instance of the model in time \( t_0 \), where \( t_0 \) is the time when the model was checked for the last time. This comparison can take many different forms. Usually, all the artefacts under consideration are expressed in a common representation, such as XML and then their \( \text{diff} \) is calculated. Change detection can take place either in predefined times (for example when a model is saved) or when the user requires to ensure that the trace link integrity is not violated. If model changes are detected, the trace links, which refer to those models should be updated. If there are predefined policies associated with the detected changes, then the link maintenance can be done automatically. State-based approaches are often limited in detecting only syntactic model changes, while semantic model changes can jeopardise the link integrity as well.

3. REQUIREMENTS FOR TRACEABILITY MAINTENANCE

If we consider the key characteristic of link integrity and traceability maintenance as discussed above, we can derive a set of requirements for a traceability maintenance approach. These requirements can be more appropriately viewed as highly desirable characteristics of a suitable approach, since the availability of even a portion of them should yield substantial benefits:

1. (R1) Detection and Correction of Dangling Links: a traceability maintenance approach should be able not only to detect dangling links (i.e. links whose integrity is compromised), but it should also be able to reestablish the link integrity. In the ideal case, the approach should be able to reestablish the compromised links automatically, i.e. without any input from the user. In the cases, where this is not possible (for example with some types of imposed links), the approach should give guidance to the user as to how to reconcile the link. This could be achieved by reducing the space of possible link end candidates. In the worst case scenario, when guidance for the reestablishment of the link integrity can not be provided, then all dangling links should at least be reported back to the user for manual reconciliation. The purpose of a good traceability maintenance approach should be to maximize automatic reconciliation and minimize the manual one.

2. (R2) Support for both imposed and inferred link maintenance: As discussed in section 2.1, there are two generic types of links. A traceability maintenance approach should be able to manage both of those link types.

3. (R3) Support for different model change types: In section 2.2, three atomic model change types have been identified. A traceability maintenance approach should be able to detect dangling links caused by all three types, namely deletion, addition and modification. A note should be made at this point concerning the addition model change type. If addition is not a part of a chain of changes then it does not compromise link integrity. However it compromises the completeness of the link set and hence an appropriate trace link identification approach should be applied. In the case where addition is a part of a chain of changes (for example a model element replacement) then the traceability maintenance approach should be able to detect possible dangling links caused by this chain of changes.

4. (R4) Support for heterogeneous notations: With the advent of DSLs, different and heterogeneous notations could be possibly used during software development. A traceability maintenance approach should not restrict users as to which notation they will use or in which ways they will evolve their models. It should be able to maintain link integrity between models represented in heterogeneous notations no matter how these models change.

Based on this set of requirements, we have developed a traceability maintenance approach. This approach is illustrated in the next section.

4. TRACEABILITY EVOLUTION WITH TML

The proposed traceability maintenance approach is part of a holistic approach to traceability presented in [9]. At the crux of this approach lies the Traceability Metamodelling Language (TML), which is a domain-specific metamodelling language [22] for traceability. The purpose of TML is to enable the construction and maintenance of traceability metamodels as well as their accompanying correctness constraints. Since the proposed traceability maintenance approach is part of TML a brief discussion of it will enhance the readability of this paper.

4.1 Traceability Metamodelling Language.

A fundamental assumption of TML is that in the general case one needs to establish trace links between elements belonging to a number of models that potentially conform to diverse metamodels. Additionally, several types of trace links linking different types of model elements may need to be captured, depending on the traceability scenario. Finally, correctness constraints that extend beyond simple type conformance should be captured. These constraints usually are domain and/or case-specific. As a consequence, the traceability engineer might need to specify multiple traceability models which conform to different traceability metamodels. The purpose of TML is to enable the construction and maintenance of traceability metamodels and accompanying constraints with reduced effort by encoding constructs and relationships that are often encountered when constructing traceability metamodels into first-class metamodelling artefacts. Moreover, TML can be used for generating the ap-
appropriate supporting infrastructure of the generated metamodels. A conceptual diagram of the holistic approach to traceability proposed by TML is illustrated in figure 1.

To make clear how TML works, consider that we need to capture traceability information in a traceability model \((T_M)\) between model \(M_A\) which conforms to metamodel \(MM_A\) and model \(M_B\) which conforms to metamodel \(MM_B\). This scenario is illustrated in Figure 2. In TML, the traceability engineer should define a TML model which captures all the possible traceability information between \(MM_A\) and \(MM_B\). This information can be case-specific trace link types, link rationale, etc. When the TML model is constructed, a number of artefacts can be generated automatically. First of all, we can generate an ECore metamodel to which the traceability model between \(M_A\) and \(M_B\) \((T_M)\) will conform to. Furthermore, a set of validity constraints, which complements the ECore metamodel, is generated.

Figure 1: Conceptual diagram of the TML Approach

In addition to the traceability metamodel and its accompanying constraints, other supporting infrastructure can be generated from a TML model. Concrete textual syntax for the case-specific traceability language between \(MM_A\) and \(MM_B\) can be generated. This is achieved by utilizing the EMFText plugin for Eclipse [1]. As a consequence, we can get for free all the additional features that come with EMFText such as a text editor. Moreover, a traceability identification script is generated. This script can be used to identify potential trace links between the two models, namely \(MM_A\) and \(MM_B\) and it can then populate the traceability model \(T_M\). Finally, a traceability maintenance script can be generated. This script can be used to reconcile dangling links in \((T_M)\), while \(M_A\) and \(M_B\) evolve. A more detailed description of TML is out of the scope of this paper. In the next section, we will illustrate how the traceability maintenance script is generated and how it can be used.

4.2 State-Based Traceability Maintenance in TML

The core of the TML metamodel as well as the classes associated with traceability maintenance are illustrated in Figure 3. Those classes are the \(MaintenanceData\) class, the \(ReconciliationExpression\) class, the \(FuzzyMatching\) abstract class and the two derived classes, namely \(Simmetrics\) and \(Wordnet\). Finally, the enumeration \(MaintenanceDataType\) defines the valid types of the type attribute of the \(MaintenanceData\) meta-class.

Figure 2: Exemplar of a Traceability Scenario

In a TML model, every \(TraceLinkEnd\) can have 0 to \(n\) maintenance data. By maintenance data, we mean data which represent particular characteristics of the model element to which a link end refers. For example maintenance data could be a simple name attribute or more complex data structures as we will demonstrate in section 5. The rationale behind capturing maintenance data is that there are some aspects of an element which justify the existence of a link. If these change then the validity of the link comes under question. These aspects are domain specific and the domain expert could capture them in the TML model. Every \(MaintenanceData\) of a link end is associated with a reconciliation expression, which is an expression in a model query language and it can access the associated data of the model element.

Figure 3: Slice of the TML Metamodel
Initially it is used to assign a value to the MaintenanceData and then it is used in the traceability maintenance script to retrieve the relevant model element data, so that it can be compared with the value of the maintenance data of the link end.

The proposed traceability maintenance approach works in the following way. First, the traceability engineer defines the maintenance data for every link end in the TML model. Additionally, the associated reconciliation expressions are defined. When the TML model is defined, then the various artefacts described in section 4.1 are generated. One of them is the traceability maintenance script. This script can be used to detect and reconcile dangling links. To detect a dangling link, the script traverses all the link ends in the TML model and compares the value of the maintenance data of each of them with the value derived from the relevant model element to which the link end refers. To derive the value from the model element the reconciliation expression which is associated to the maintenance data is used. When a discrepancy is detected then this link is flagged as dangling.

When a link is flagged as dangling, then the maintenance script attempts to reconcile it. If this is not possible, then the user is notified of the suspect link. The process for reconciling a dangling link is illustrated in Figure 4. For a dangling link end, the maintenance script tries to find a model element whose derived value can match the maintenance data of the link end. Initially the script looks for a matching model element in the model to which the link end was referring in the first place. If no such match is found, then the script looks for an element with the required characteristics in the rest of the models in the workspace. If the exact match is found (for example one could have deleted and then recreated the model element) then the link end is reconciled automatically. In the case where more than one exact matches are found, user input is required as to which of the matches should be referenced by the link end. If no exact matches are found by the maintenance script, then fuzzy matching algorithms could be used. For the time being we have integrated in the current implementation of TML two tools for fuzzy matching, Simmetrics[4] and Wordnet. However, the implementation is flexible and the traceability engineer can implement an appropriate fuzzy matching tool based on her needs. In the case where, one match is found using the fuzzy matching approach, then the script checks whether the similarity is above a similarity threshold, which is specified in the TML Model. If it is above this threshold, then the link end is reconciled automatically. If not, user input is required. If more than one matches are found, then all the matches are reported back to the user sorted based on their similarity and user input is requested. Finally, in the case where no matches can be found, the link end is flagged as dangling and it is reported back to the user for manual reconciliation.

The traceability maintenance script can run either manually when the traceability engineer wishes to reestablish the integrity of the traceability model or it could run automatically when model changes are detected. The latter is implemented by coupling the proposed approach with Concordance [19], which is an index to store and manage cross-model references and metamodel usage. Concordance provides a cross-model reference reconciliation client, which can detect changes incurred at the model level. A trace maintenance script can be attached to a particular model and when the client detects that the model has changed, the script can be used to reconcile the problematic links.

The current implementation of our approach is build using the Epsilon framework [11]. The traceability script as well as the reconciliation expressions are written in the Epsilon Object Language (EOL) [14], which is at the core of the Epsilon framework. EOL is an imperative programming language for creating, querying and modifying EMF models. To generate the maintenance scripts from the TML models we have employed, a template-based model-to-text transformational approach using EGL [20] as the template language. Any model-to-text language such as MOFScript [16] or XPand [12] could have been used instead.

The approach to traceability maintenance which is proposed in this paper satisfies the requirements identified in section 3 in the following way:

1. **(R1) Detection and Correction of Broken Links:** the approach is capable of not only detecting but also of reconciling broken links. The success of the approach relies on how well and precisely the traceability engineer captures the required maintenance data for a given link end.

2. **(R2) Support link maintenance for both imposed and inferred links:** Since the rationale for a link is captured in the TML model by the model engineer, the approach does not rely on an explicit relationship between model elements for reconciling the trace links. Therefore, it supports both imposed and inferred link maintenance.

3. **(R3) Support link maintenance for all model change types:** Since the proposed approach does not rely on identifying model changes, but only on different states of the models under consideration, it can support different types of model changes.

4. **(R4) Support for heterogeneous notations:** TML is a generic approach to traceability. All the required traceability information are captured in the TML model from which the maintenance script is generated. Therefore, for every set of notations a TML model can be defined. The only constraint of TML is that the metamodels of the various notations should be expressed in a common meta-metamodel. In the current implementation this is ECore, which is the metamodel of the Eclipse Modeling Framework [10].

In the next section, we will present the practicality of our approach using a simple but representative example.

## 5. **EXAMPLE**

In this section, we give a concise example of how to maintain trace link integrity between two models. The aim of this example is to demonstrate how traceability information can be maintained between a component model and a class model. Such a scenario can arise in a Component-Based Development Environment, where class diagrams are used to refine the architecture specified by component diagrams into a concrete design. In our example, we use two simple metamodels, the ClassMetamodel and the ComponentMetamodel, which are illustrated in Figure 5 respectively. Our aim is to maintain traceability links between models which
conform to those two metamodels, i.e. the ComponentModel and the ClassModel. More precisely, we want to maintain trace link integrity for links which capture traceability between instances of Package from the ClassMetamodel and instances of Component from the ComponentMetamodel, while those instances evolve. We assume that an initial set of trace links has been captured and represented in a TML model.

The first step of our solution is to capture maintenance data for the link ends of the link we want to maintain, which in our case is the link between instances of the Package meta-class and instances of the Component meta-class. We assume that the validity of this link can be jeopardised if we change the name of the package, if we change the name of the component or finally if we change the classes a package contains. Hence, we need to define three different types of maintenance data. Namely, we define the PackageNameData and the ClassContainmentData for the PackageTraceLinkEnd and the ComponentNameData for the ComponentTraceLinkEnd. To retrieve the values of these attributes, we define two reconciliation expressions, namely the NameExpression and the PackageContainmentExpression. The former is used for retrieving the Package and Component name, while the latter is used for retrieving the Classes contained in a Package. For the PackageNameData and the ComponentNameData, we will use the WordNet tool and the Simmetrics tool for supporting fuzzy matching. For the Simmetrics tool we will use the Hamming Distance for comparing the signature that will be produced by the NameExpression with the names of the various packages and components during the reconciliation process. Apart from the Hamming Distance, all the different similarity measures provided by the Simmetrics tool are supported by TML. The TML model for this example is illustrated in Figure 6.

Both of the reconciliation expressions are written in EOL [14] and they are illustrated in Figure 7. Although the NameExpression is simple, the PackageContainmentExpression is more complex. The NameExpression will retrieve the name value from instances of the Package or of the Component meta-classes. The PackageContainmentExpression traverses all the contents of a package, finds the Classes which are contained in that Package and creates a String signature for the Package. As it is obvious complex expressions can be used as reconciliation expressions and their complexity is only limited by the expressiveness of the query language that is used.

After specifying the TML model we can generate a maintenance script which can be used to maintain all instances of the two metamodels. If we need to evolve the traceability

---

**Figure 4: Process for the Reconciliation of Dangling Links**

**Figure 5: ClassMetamodel and ComponentMetamodel**

The first step of our solution is to capture maintenance data for the link ends of the link we want to maintain, which in our case is the link between instances of the Package meta-class and instances of the Component meta-class. We assume that the validity of this link can be jeopardised if we change the name of the package, if we change the name of the component or finally if we change the classes a package contains. Hence, we need to define three different types of maintenance data. Namely, we define the PackageNameData and the ClassContainmentData for the PackageTraceLinkEnd and the ComponentNameData for the ComponentTraceLinkEnd. To retrieve the values of these attributes, we define two reconciliation expressions, namely the NameExpression and the PackageContainmentExpression. The former is used for retrieving the Package and Component name, while the latter is used for retrieving the Classes contained in a Package. For the PackageNameData and the ComponentNameData, we will use the WordNet tool and the Simmetrics tool for supporting fuzzy matching. For the Simmetrics tool we will use the Hamming Distance for comparing the signature that will be produced by the NameExpression with the names of the various packages and components during the reconciliation process. Apart from the Hamming Distance, all the different similarity measures provided by the Simmetrics tool are supported by TML. The TML model for this example is illustrated in Figure 6.

Both of the reconciliation expressions are written in EOL [14] and they are illustrated in Figure 7. Although the NameExpression is simple, the PackageContainmentExpression is more complex. The NameExpression will retrieve the name value from instances of the Package or of the Component meta-classes. The PackageContainmentExpression traverses all the contents of a package, finds the Classes which are contained in that Package and creates a String signature for the Package. As it is obvious complex expressions can be used as reconciliation expressions and their complexity is only limited by the expressiveness of the query language that is used.

After specifying the TML model we can generate a maintenance script which can be used to maintain all instances of the two metamodels. If we need to evolve the traceability

---

28
A similar approach to traceMaintainer is the one proposed by Murta et al [15]. ArchTrace is a tool that addresses the consistency and evolution of trace links between software architecture models and their associated code. ArchTrace relies on an infrastructure for identifying change events in the architecture models and continuously updating the trace links based on the change events and a set of policies. These policies are atomic elements which can be disabled and enabled individually so that they fit to the user’s situational needs. The ArchTrace tool assumes the use of xADL 2.0 [7] to describe software architectures and Subversion [6] to store source code.

A state-based approach is proposed by Sharif and Maletic [21]. In this approach a difference tool such as EMFCompare is used to identify syntactic differences between different versions of a model. Based on these differences and user input the links are evolved. Maletic et al. [13] propose an analogous approach. Text differencing is used to identify syntactic changes in different versions of source code or of XML representations and according to the identified changes user input is required to reconcile the dangling trace links.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented an approach to maintaining the validity of trace links between models expressed in different modeling languages. This validity is compromised due to the inevitable evolution of the models to which a trace link refers. We have proposed a generic state-based solution which relies on capturing link end specific meta-data as well as expressions to retrieve them. This metadata can be used to automatically generate maintenance scripts which in turn can be used for the evolution of the traceability models.

In the future, we plan to apply the proposed approach in a large and complex case study in order to investigate its applicability in such scenario. Using this case study, we plan to carry out an empirical study on how much extra effort is required to define the maintenance data and reconciliation expressions for each trace link. Moreover, we would like to investigate how much time is gained by using the proposed approach versus the time needed to maintain traceability information manually. Finally, in the future we plan to investigate possible extensions to the proposed method to cover traceability scenarios which include textual artefacts in addition to models.

8. REFERENCES


Resolving Feature Dependency Implementations Inconsistencies during Product Derivation

Saad bin Abid
Lero- The Irish Software Engineering Research Centre
University of Limerick, Limerick, Ireland
Saad.binabid@lero.ie

ABSTRACT
Features implementing the functionality in a software product line (SPL) often interact and depend on each other. It is hard to maintain the consistency between feature dependencies on the model level and the actual implementation over time, resulting in inconsistency during product derivation. We describe our initial results when working with feature dependency implementations and the related inconsistencies in actual code. Our aim is to improve consistency checking during product derivation. We have provided tool support for maintaining consistency between feature dependency implementations on both model and code levels in a product line. The tool chain supports the consistency checking on both the domain engineering and the application levels between actual code and models. We report our experience of managing feature dependency consistency in the context of an existing scientific calculator product line.

Categories and Subject Descriptors
D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

General Terms
Management, Performance, Experimentation, Languages, Verification.

Keywords
Software product lines, aspect-oriented product line, AspectJ programming, feature implementation dependencies, variability models, product derivation, consistency checking, tool support.

1. INTRODUCTION
A software product line (SPL) is a “set of software intensive systems sharing a common, managed set of features that satisfy the specific market segment or mission and that are developed from a common set of core assets” [1]. Using an SPL approach allows companies to realize significant improvements in time-to-market, cost, productivity, and system quality [2]. SPL engineering consists of two main interlaced activities, domain engineering and application engineering. During domain engineering, the commonalities and the variation points of the product family are analyzed and maintained. Variability points indicate where there will be variations in the same product family. During application engineering the variation points identified during domain engineering are realized based on a given configuration and a concrete product is derived (Product derivation). One major difficulty with SPL engineering is to deal with thousands of variation points in an industrial size product line [25]. These variation points need special attention as they add complexity during product configuration (PC) and product derivation (PD).

Variability management can greatly impact the complexity that is involved when producing a new product from existing product line assets [3]. The products of a SPL family differ by the features they include in Feature-Oriented Domain Analysis (FODA) [4,5]. SPL should have the capability to allow configuration of features or addition of individual features to the system. It is common to find cross cutting variable features during FODA. Cross cutting features are the features whose functionality spans over several parts of an application. Cross cutting variability makes it difficult to map features to architectural design and then to implement these variabilities in source code. Software product line models are inherently complex in nature due to embedded variability. Maintaining the consistency of product line artefacts which is necessary for derivation of correct products is a challenge for product line engineers in practice.

Aspect-oriented programming (AOP) [6] is a paradigm which allows developers to capture cross cutting structure and cross cutting concerns in a modular fashion. It helps to modularize feature implementations in source code. The common features are implemented in a base structure and variable features are implemented as aspects. During aspect oriented product line engineering (AOPLE) [17] product derivation, an aspect weaver creates a product by weaving variable feature implementation (aspect) into the base structure. Several approaches [7,8,9] using AOP have been conducted to implement cross cutting features in a modular fashion using separated components called aspects.

Features are not in general independent of each other. Changes in the implementation of one feature will cause side effects in the implementation of other features [10]. The problem is caused due to the fact that feature dependencies are embedded into feature implementations, resulting in tangled code issues. In feature implementations it is possible that feature dependencies are, 1) not correctly implemented, 2) missing, 3) wrongly implemented. Such issues can cause problems during product derivation when an aspect weaver creates a product by weaving aspects into the base modular structure.

In this paper, we provide a tool suite based on the Eclipse Modeling Framework (EMF) [11] and Epsilon [12] languages to 1) automatically extract feature implementation dependencies, 2)
Activation dependency aspects can be classified into four different categories namely, 1) Functionality between functional features describing that one represents the modification behavior from F1 to F2. We can see [13].

Activation dependencies aspects can be seen as separate generic aspects. Activation dependencies aspects that implement the requirement that both the features must be active simultaneously. In Scicalc-PL the aspectual implementation artefact contains Java classes and Aspects. The dynamic cross cutting mechanisms (i.e., Pointcut and Advice) of AspectJ are used to extend one feature’s interaction part with other functional features functionality which actually implements the modification dependency. The activation dependencies are implemented using generic aspects.

The focus of this work is to 1) automatically generate and visualize cross cutting feature dependencies in an existing implementation model (AML model), 2) check for consistency of model-to-code dependency aspects and interactively resolve any inconsistency in between, 3) check the consistency of a given configured AML model dependency aspects with respect to the corresponding implementation code base dependency aspects.

2. BACKGROUND

Variability management is a key to success for any SPL. Many SPL research approaches focus on single development artefacts consisting of isolated models (e.g. feature-oriented, component-based product derivation etc.) [14,15]. While viewing SPL as a collection of artefacts has many advantages, there are disadvantages as well including the problem of describing cross cutting features, mapping cross cutting features to architectural design and then implementing them in the source code. In order to exploit the real benefits of a product line we need to connect these isolated artefacts (i.e., models, implementation code, and documentation).

The approach represented in this paper is illustrated using the example calculator product line (Scicalc-PL) artefacts presented in [10,13]. In this paper we are focusing on the implementation model (AML model) and the implementation code base artefacts of the Scicalc-PL. The AML model is an abstract view on the actual implementation consisting of packages, Java classes, Aspects and dependency relationships among them. The dependencies between the cross cutting features are implemented as Aspects in the source code of Scicalc-PL. The process of how the variabilities and dependencies are analyzed and utilized for structuring the product line implementation is discussed in [13].

The work in [13] focuses on two different kinds of dependency aspects which implement functional feature dependencies at runtime, namely Modification and Activation Dependencies [16]. For the clarification of the reader we discuss each briefly with an example dependency aspect from feature F1 to feature F2.

Modification dependency from F1 to F2 implies that functional behavior related to F1 can be divided into two parts: the functional core and the interaction. The functional core represents the main functionality of the feature F1. The interaction part represents the modification behavior from F1 to F2. We can implement the interaction part (i.e., the modification dependency) of functional feature F1 using an AspectJ aspect (for more detail see [13]). Activation dependencies affect the activation of functional features. Activation dependencies can be implemented as separate generic aspects. Activation dependencies aspects can be classified into four different categories namely, 1) Excluded Activation dependency aspects: aspects that implement the functionality that during execution the functionality of one functional feature can only be activated if the functionality of the other feature is active (in other words functionality of one requires the other), 2) Sequential Activation dependency aspects: aspects that implement the requirement that both the features must be active sequentially, 3) Concurrent Activation Dependency aspects: aspects that implements the requirement that features must be active together concurrently.

In Scicalc-PL the aspectual implementation artefact contains Java classes and Aspects. The dynamic cross cutting mechanisms (i.e., Pointcut and Advice) of AspectJ are used to extend one feature’s interaction part with other functional features functionality which actually implements the modification dependency. The activation dependencies are implemented using generic aspects.

The focus of this work is to 1) automatically generate and visualize cross cutting feature dependencies in an existing implementation model (AML model), 2) check for consistency of model-to-code dependency aspects and interactively resolve any inconsistency in between, 3) check the consistency of a given configured AML model dependency aspects with respect to the corresponding implementation code base dependency aspects.

3. INCONSISTENCY BETWEEN DEPENDENCY ASPECTS DURING PRODUCT DERIVATION

In order to identify potential challenges for managing dependency aspects consistency, we analyzed the example Scicalc-PL model-based product derivation. The inconsistency scenarios defined in this section are in the context of research work conducted in [13].

We believe that the following inconsistency scenarios have to be dealt with in order to manage consistency in any AOPLE [17]. These are not the only type of inconsistency scenarios that can occur but for this particular stage of our research we have considered only the following inconsistency scenarios with scientific calculator example (Fig.1). As a running example we will demonstrate the inconsistency scenarios using dependency aspect examples taken from Scicalc-PL implementation model (Fig.2).

Figure 1. Calculator Application [10]
In Figure 2, modification dependency is implemented by an aspect. <<Feature>> NumberSystems modifies the functionality of <<Feature>> History during feature interaction. <<HistoryModule>> and <<NumberSystemmodule>> implement the functionality of the History and NumberSystems features respectively. Both the modules consist of core functionality and interaction functionality.

The Aspect HistoryModificationNumberSystem uses the methods in <<Historymodule>> and <<NumberSystemmodule>>. The following sections shall describe not only modification dependency inconsistency but also the other types of dependency aspects discussed in Section 2.

3.1 Dependency Aspects Missing

During our analysis of Scicalc-PL model-based product derivation, we observed that inconsistency occurs when feature dependencies are 1) not implemented in source code, 2) not included and synchronized with the implementation model (abstract view on code), 3) not configured during product configuration. An inconsistency can occur when, for instance, 1) the Aspect HistoryModificationNumberSystem is not implemented in the scicale-PL code, 2) if it is not implemented in AML model on domain engineering, 3) not included in configured implementation model during product derivation. This leads to an inconsistency which won’t allow the features History and NumberSystems to work in the final executable product. The same applies to other types of activation dependencies briefly discussed in Section 2, as all the other types of feature dependencies are implemented using aspects.

3.2 Implementing the Wrong Feature Dependency Aspects

An inconsistency can occur when dependency aspects are implementing the wrong feature dependencies. For instance, in the above mentioned example if the aspect HistoryModificationNumberSystems advises the method in <<NumberSystemsmodule>> and not the method in <<HistoryModule>>.

Another example scenario can be if the feature History has Excluded Activation dependency with the feature NumberSystems(History—Excludes-Activation-dependency  NumberSystems). Which means that the History functionality should be excluded when NumberSystems feature is turned on in the final product. If the situation is like NumberSystems—Excludes-Activation-dependency  History, then it leads to inconsistency in the final executable product. The mentioned inconsistency situation needs to be resolved during product derivation Otherwise the final product may not be robust and fully functional. This applies to other types of feature dependency aspects as discussed in Section 2.

3.3 Partially Implemented Feature Dependency Aspects

We analyzed Scicalc-PL and found that partially implemented feature dependencies also contribute to raising inconsistency during product derivation. An example scenario can be when the feature History has a modification dependency with the feature NumberSystem in the interaction part of the History Class. If there is some other method in History that needs to be modified during feature interaction, which is not yet implemented, this leads to a partially implemented feature dependency aspect. During model-based product derivation, it is possible that such modification is not implemented at all or not included in the implementation model (AML model). In both cases this leads to an inconsistency during product derivation.

3.4 Order not Maintained or Implemented in Feature Dependency Aspects

In Scicalc-PL, there are features which need to be activated sequentially. For instance, the feature Angle has Sequential Activation dependency with the feature Display in the Scicalc-PL implementation [10]. Both have to activate in sequence in order to produce a robust and fully functional product during product derivation. The aspect implementing Sequential activation dependency must take into consideration the activation of features having sequential dependency. The implementation model must also implement this functionality so that when the implementation model is configured for a particular product this feature dependency gets included. This feature dependency aspect order inconsistency may or may not apply to other types of feature dependency aspects.

3.5 Inclusion of Redundant Feature Dependency Aspects

Introduction of redundant feature dependency aspects in the final list of components/configured implementation model may decrease the efficiency of product derivation and can increase the product derivation time. Implementation of redundant feature dependency aspects may cause inconsistencies in large scale product lines where there are thousands of cross cutting variation points.

4. ASPECTUAL CONSISTENCY MANAGEMENT APPROACH

In the previous section, we identified some of the many inconsistency scenarios which can, 1) force a product line into an inconsistent state, 2) cause inconsistency issues during model-
based product derivation and product configuration, 3) produce a faulty product with lesser or erroneous functionality.

Figure 3 represents how our work is situated in the context of the overall Scicalc-PL case study [10]. We developed a plug-in tool suite chain along with the graphical representation of models for checking inconsistencies in feature dependency aspects. We have used the following frameworks and languages for developing our tool suite:

- The EMF incremental plug-in development environment for tool suite plug-in development
- Eugenia, the Epsilon framework graphical language for developing graphical editor for Scicalc-PL model artefacts.
- Epsilon framework validation language (EVL) for applying constraints on models.
- EMF Compare language [18] to compare models.

The following sections will discuss our approach in detail.

4.1 Extracting Feature Dependency Relationships from Implementations

During analysis of Scicalc-PL, we identified certain inconsistencies (addressed in Section 2) that can occur while deriving a product if the implementation model (A) is not synchronized with actual implementation code base (B). See Figure 3. For this purpose we developed two EMF based plug-ins. See (1) in Figure 3. **Code2Aml plug-in** parses the implementation and creates an implementation model without taking feature dependency aspect relationships into account. Scicalc-PL implementation project is an AspectJ project [19]. In order to extract the feature dependency relationships, we developed another plug-in which actually works with aspects implementing feature dependencies in the Scicalc-PL implementation project. The AspectJ development project maintains an abstract syntax tree (AST) for the project. The **Aspectjrelmapbuilder plug-in (1b)** traverses the actual AspectJ AST relationship map of Scicalc-PL implementation to find out relationships between the Java classes and aspects. The plug-in (1b) than automatically generates the implemented feature dependency relationships in the already generated implementation model (AML model) using the **code2aml** plug-in. Both the plug-ins (1) are resource change sensitive. It means that whenever the Scicalc-PL AspectJ project changes (addition/deletion/update of Java classes and aspects implementing feature dependencies), both plug-ins get activated in the background.

4.2 Managing Feature Dependency Aspects

During analysis of Scicalc-PL artefacts, we found that there is a need of maintaining feature dependency aspect relationships. It is because feature dependency implementation relationships in the implementation need to be synchronized with the implementation.
model at any development stage. To solve the mentioned challenge of maintaining feature dependency relationships, we established **AspectJrelMap** meta-model represented in Figure 4. The proposed meta-model also acts as a traceability model between the implementation model (AML model) and the actual implementation. It contains the AspectJ implementation concepts (i.e., Advices, DeclaresOn, etc.), which are used to implement features dependencies described in Section 2. Proposed **AspectJrelMap** meta-models can be instantiated for any project implementation developed in the AspectJ development environment. It maintains a snapshot of relationships between Aspects and Java Classes existing in the implementation.

**Figure 4. AspectJ relationship map meta-model**

![AspectJ relationship map meta-model](image)

**Figure 5. Example instance of AspectJrelMap meta-model**

Figure 5 presents an example initiated **AspectJrelMap** meta-model in the EMF model editor. It captures the sources and targets along with the function implementing relationship.

### 4.3 Visualizing Feature Dependencies Implementation Relationships

The graphical editor is generated using the Eugenia graphical language. Visual representation of feature dependency aspects helps us to visually identify inconsistencies in the implementation. It helps us to detect inconsistent evolutionary changes and resolve them interactively in the product line. It also helps us to identify and trace inconsistency (e.g., missing aspects, Java classes, dependency sources and targets) in the configured implementation model (C). Visualizing and resolving inconsistencies will be discussed in the following sections.

### 4.4 Applying Constraints on Feature Dependency Aspects

We have used EVL constraints to evaluate consistency during the domain engineering and application engineering phases (Figure 2). During domain engineering, when process (1) is executed we obtain an implementation model, which acts as a model artefact for the implementation. During domain engineering, after time T evolutionary changes result in a new version of the implementation model (A). We use EMF Compare language to identify potential changes in the new version of the implementation model with respect to the old implementation model (A). EVL constraints (2) are then applied to generate the diff model. Based on the failed constraints, the error markers are generated in the new version of the implementation model (A). During application engineering, the product line engineer wants to find out if the given configuration is consistent and includes all the required feature dependencies. In order to check the consistency of the configured implementation model, process (2) is again executed and inconsistencies are marked with error markers in the editor.

We describe our consistency rules into two levels, namely completeness constraints and dependency implementation constraints.

#### 4.4.1 Completeness constraints

These consistency checking rules check for the completeness of the generated implementation model (AML model). After the introduction of evolutionary changes, the new version of the implementation model is checked for completeness consistency. Table 1 shows a few of the completeness constraint types applied on the generated AML model. These constraints also apply to the situation when a configured implementation model needs to be checked for completeness.

#### 4.4.2 Dependency implementation constraints

These consistency checking rules are applicable to generated feature dependency relationships. These constraints check for generated dependency relationships attributes like name, sources
and targets of the aspect implementing feature dependency. Table 1 shows a few of the dependency implementation constraints.

<table>
<thead>
<tr>
<th>Completeness constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name and qualified name of model element (Java class/Aspect/Package/Project) defined in extracted implementation model</td>
</tr>
<tr>
<td>Evolutionary change (addition/deletion/update) implemented in extracted implementation model</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dependency implementation constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name and qualified name of aspect implementing the feature dependency in implementation synchronized with model element representing the aspect in extracted implementation model</td>
</tr>
<tr>
<td>New dependency implementation in actual implementation is implemented in extracted implementation model</td>
</tr>
<tr>
<td>Function implementing feature dependency is present in implementation model (aspect in model) and in synchronous with actual implementation (aspect actual implementation)</td>
</tr>
<tr>
<td>Sources and targets of aspects implementing feature dependency is in synchronous with sources and targets in extracted implementation model</td>
</tr>
</tbody>
</table>

Table 1. Completeness and dependency implementation constraints

4.5 Feedback to Product Engineer during Product Derivation

After applying the constraints using EVL the product engineer gets feedback in the form of error markers. These error markers help the product engineer to identify the potential inconsistencies to be resolved during product derivation. The inconsistency resolution is dealt with by suggesting the fixes in the “Quick fixes” section. The resolution choices are provided purely on the basis of implemented feature dependency aspects in actual implementation. Fig. 6 represents a scenario where target of an aspect is changed and on the basis of the pre-existing implementation model we provide choices (not shown in the Fig. 6) to the product engineer to resolve the inconsistency. Figure 6 (1) represents the Eugenia based graphical editor and (2) represents the error view.

4.6 Interactive Resolution of Feature Dependency Implementation Inconsistencies

The graphical editor developed in Eugenia is helpful in interactively resolving the inconsistencies. For example when an inconsistency arises, an error marker is generated on the respective Aspects, Java Classes or Packages. Double clicking the error detail in error view of the editor takes the product engineer to the problematic graphical element (Package, Java Class or Aspect).

5. DISCUSSION

In this paper we have presented our initial results obtained when working with feature dependency implementations in the context of product derivation in a product line. We used a case study product line to analyse the overall problem of inconsistency and to evaluate our approach. In this paper, we introduced our experimental approach for extracting and maintaining feature dependency constraints.
dependency implementations.

We began by developing an incremental plug-in which actually reverse engineers the existing implementation and generates an implementation model (AML model). In order to generate the feature dependency implementation relationships, we obtain the AspectJ abstract syntax tree (AST) and work with the relationship map to find out relationships between implementation elements (e.g., Java Classes, Aspects). This information is then used to generate the relationships between extracted elements of the implementation model. We have also proposed a meta-model for capturing the AspectJ project relationship map information. The meta-model also acts as a traceability model between the generated implementation model and the actual code in the way that it links the relationship map to the AspectJ project implementation. In order to synchronize the generated implementation model and actual implementation, we use EVL constraints which we apply on the generated model for checking completeness and dependency implementation consistency. During product derivation when product engineer wants to check if the configured implementation model (AML model) contains all necessary feature dependency implementations, we input the configured model along with implementation model representing the actual implementation to (2) (Fig.3). As an output the product engineer may or may not get a list of errors which he can work on to resolve them interactively.

To facilitate the product engineer to identify inconsistencies we developed a graphical editor. The inconsistencies are shown as graphical markers in the Eugenia based editor which can help the product engineer to visually see the inconsistencies. In order to resolve the inconsistencies, we extended the quick fix capability of the EMF validation framework using EVL. The developed prototype tool suite has some limitations. For this particular stage of the research work, we are assuming that all the feature dependencies are present in the implementation. When a new dependency is generated in the implementation model it represents the feature dependency in the feature model in domain engineering (Fig 3). The actual Scicale-PL implementation consists of 348 different relationships. We obtained this information by analyzing the abstract syntax tree maintained by AspectJ development environment for Scicale-PL implementation project. We obtained all the relationships and generated them in the implementation model successfully. The feature dependencies in Scicale-PL were implemented with four main AspectJ programming functions (i.e., Advises, Advised by, Declared on and Aspect declarations). During application engineering, we are not taking into consideration how the configuration process is performed, rather we are checking consistency with respect to a given product configuration/AML model. We are currently completing our research prototype in order to include the feature model so that we can actually check the inconsistencies from domain engineering to the application engineering process.

6. RELATED WORK
Automated tool support is highly desirable for managing the complexity and variability inherent in software product lines.

Work by Lienhard et al.[20] analyzes the problem of runtime dependencies between features in an object-oriented system. It provides the detection strategy based on meta-models which capture the references. It also provides a visualization of feature runtime dependencies.

Kothari et al. [21] proposed an approach to system comprehension that considers features as the primary unit of analysis. The work provides a mechanism to define a relationship between features based on comparing the feature implementation.

Work in [22] presents a technique for semantic conflict detection between aspects at shared join points. The approach is based on abstracting the behavior of advice to a resource-operation model, and detecting conflicts patterns within the sequence of operations that various aspects apply to each resource.

Recent work by Vierhauser et al. [23] applies tool support for incremental consistency checking on variability models in the industrial case study. However the approach is not taking feature dependencies into consideration and works on two variability models, assets and decisions.

Our approach differs from the above mentioned approaches in that we are allowing the product engineer to interact and resolve the inconsistencies. We also provide tool support to identify inconsistencies and interactively resolve them during domain engineering (maintaining and synchronizing implementation model and actual implementation) and in application engineering (consistency checking the configured implementation model with respect to actual implementation).

7. CONCLUSION
Our primary goal is to make product derivation less error prone and more efficient. To achieve efficient product derivation we need to manage inconsistencies at all development stages in SPLs, particularly during product derivation. In this paper we have identified some of the different types of inconsistencies affecting feature dependency implementation that can cause product derivation to be inefficient and error prone. The base work for this research is described and elaborated in [10,13]. To support the product engineers in handling inconsistencies in large SPLs we have provided a prototype tool suite and an approach for managing feature dependency implementation inconsistencies. The tool suite has the capability to 1) extract the implementation model and generate dependencies from an actual implementation, 2) facilitate the product engineer to synchronize the implementation and the implementation model by applying EVL constraints, 3) detect and resolve the inconsistencies in the given configured implementation model, 4) provide the visual identification and interactive resolution of inconsistencies in the implementation model representing actual implementation and the configured implementation model for a particular product. In future we are planning to improve our approach by identifying and implementing the remaining feature dependency inconsistency scenarios. We are also planning to include the remaining Scicale-PL modeled artefacts shown in Fig 3 (i.e., Feature and FIM models) in our approach. It is also planned to improve the inconsistency checking by applying incremental inconsistency checking [24] and better visualization to resolve inconsistencies during product derivation.

8. ACKNOWLEDGEMENT
This work was supported by Science Foundation Ireland grant 03/CE2/I03_1 to Lero- the Irish Software Engineering Research Centre (www.lero.ie). I would also like to acknowledge Dr. Norah Power and Dr. Goetz Botterweck for their feedback and support while preparing this research work.
9. REFERENCES


[24] Egyed, A. 2006. Instant Consistency Checking for the UML. In 28th International Conference on Software Engineering (ICSE06), Shanghai, China

Tool Support for Generation and Validation of Traces between Requirements and Architecture

Arda Goknil
Software Engineering Group, University of Twente
7500AE, Enschede, the Netherlands
a.goknil@ewi.utwente.nl

Ivan Kurtev
Software Engineering Group, University of Twente
7500AE, Enschede, the Netherlands
kurtev@ewi.utwente.nl

Klaas van den Berg
Software Engineering Group, University of Twente
7500AE, Enschede, the Netherlands
k.g.vandenberg@ewi.utwente.nl

ABSTRACT
Traceability is considered crucial for establishing and maintaining consistency between software development artifacts. Although considerable research has been devoted to relating requirements and design artifacts with source code, less attention has been paid to relating requirements with architecture by using well-defined semantics of traces. We present a tool that provides trace establishment by using semantics of traces between R&A (Requirements and Architecture). The tool provides the following: (1) generation/validation of traces by using requirements relations and/or verification of architecture, (2) generation/validation of requirements relations by using traces. The tool uses the semantics of traces together with requirements relations and verification results for generating and validating traces. It is based on model transformations in ATL and term-rewriting logic in Maude.

Categories and Subject Descriptors
D.2.1 [Requirements/Specifications]: Tools; D.2.2 [Design Tools and Techniques]: Computer-aided software engineering (CASE)

General Terms
Documentation, Design, Languages, Verification

Keywords
Tools, Traceability, Generation and Validation of Traces, Architecture Verification, Requirements and Architectural Models

1. INTRODUCTION
Traceability is considered crucial for establishing and maintaining consistency between software development artifacts such as requirements documents, architectural design, detailed design, source code and test cases. The benefits of traceability are widely acknowledged today and there are tools to record and manage trace information. Despite many advances in tools, traceability remains a widely reported problem area in industry [14]. Some traceability approaches aim at generating trace information automatically [9] [10]. Egyed et al. [10] proposes an automated traceability approach that uses a small number of traces as input. In addition to that, some heuristics helping to define the meaning of trace dependencies are proposed.

Considerable research has been devoted to relating requirements and design artifacts with source code. Less attention has been paid to relating requirements with architecture by using well-defined semantics of traces. Most approaches focus on generating traces between requirements and source code or between design and source code. In most tools and approaches, there is a lack of precise definition of traces between requirements and architecture. This lack may cause incomplete and invalid trace generation for requirements and architecture.

Traceability influences a number of software development activities such as release planning, change impact analysis, testing, and requirements reuse [8]. In this respect, these activities may produce deficient results because of invalid and incomplete traces. For instance, change impact analysis may produce high number of false positive and false negative impacted elements. Consequently, the cost of implementing a change may become several times higher than expected [8].

In this paper, we present a tool that provides trace establishment by using semantics of traces between R&A (Requirements and Architecture). The tool provides the following: (1) generation/validation of traces by using requirements relations and/or verification of architecture, (2) generation/validation of requirements relations by using traces. Generating traces is the activity of deducing traces between requirements and architecture based solely on verification of architecture and/or the requirements relations. Alternatively, the software architect can provide an initial set of traces as input. Validating traces is the activity of identifying the traces which do not obey trace semantics. Our tool checks if the relations between requirements are preserved in their implementation in the architecture. This preservation is also used in the concept of software reflexion models where relations between elements in high-level models are preserved in their implementations [24].

In the implementation of the tool, we bridge three technical spaces [19]: Semantic Web, Term rewriting, and Model-Driven Engineering. In [12], we provide tool support for consistency
checking and inferencing based on the semantics of relations for requirements. The requirements metamodel together with semantics of requirements relations in first-order logic (FOL) are given in [12]. The tool in [12] uses the semantic web technologies OWL and Jena for inferencing and consistency checking (Semantic Web technical space). The output of the tool in [12] is the requirements model which is used as an input by our tool for trace generation and validation. The architecture is expressed in Architecture Design and Analysis Language (AADL) [2]. We have defined dynamic semantics for part of AADL in terms of rewriting logic supported by the Maude language and tools [6] [7]. This enables performing simulation and verification of AADL models [29] (Term rewriting technical space). The result of the verification, which might be a counter example or execution trace, is one of the inputs to generate and validate traces. We use a trace metamodel with commonly used trace types. The semantics of the traces is provided with a formalization [13] in first-order logic. We use the formalization of traces together with requirements relations and verification results for generating and validating traces. The core part of the tool using verification results, requirements relations and traces as input is implemented as model transformations in ATL [18] (Model-Driven Engineering technical space).

The paper is structured as follows. Section 2 describes the overview. Section 3 presents the features of the tool. In Section 4, we explain the architecture of the tool. Section 5 describes the evaluation of the tool. Section 6 describes the related work, and in Section 7 we conclude the paper.

2. OVERVIEW

Our tool supports trace generation and validation with different degrees of automation. Figure 1 gives the overview of the tool with inputs and outputs. The tool takes requirements model, trace model, architecture model, and reformulated requirement as input with the constraints derived from the semantics of traces and requirements relations.

![Figure 1 Overview of the Tool](image)

The tool checks if the requirements are satisfied by the architecture. This is done by reformulating the requirements in terms of logical formulas over the architecture. To check the formulas we perform architecture simulation and verification in Maude by using input architecture model. According to the result of the verification, traces with different types (AllocatedTo or Satisfies) are generated between the reformulated requirement and the architecture. Traces are generated accordingly in the output trace model. In the validation, the tool compares the assigned traces in the input trace model with the architectural elements in the verification output. The invalid assigned traces are reported in the output error model.

The input requirements model contains the given and inferred requirements relations (see [12]). The constraints derived from the semantics of trace and requirements relations are used to deduce the new traces. Traces are generated for requirements which do not have any traces but which are related to requirements with traces. The output trace model contains the generated traces. The generation is vice versa. The same constraints are used to generate requirements relations from traces between requirements and architecture. The tool also uses the requirements relations and the constraints to check the validity of the assigned traces in the input trace model and the validity of requirements relations in the input requirements model. Invalid traces and requirements relations are reported in the output error model.

We have to note that all generated/invalid traces and requirements relations are candidates and suggestions for the software architect. They have to be checked by the architect for the final decision.

We depict the usage of the tool in a modeling process with trace generation and validation. This process is based on the analysis of activities during modeling of requirements, architecture and traces. Figure 2 gives a UML activity diagram of the process.

![Figure 2 Modeling Process with Trace Generation and Validation](image)

The process in Figure 2 consists of the following activities:

**Modeling Requirements & Designing Architecture:** This activity takes the requirements document as input and produces the input requirements model, input architecture model and input trace model for trace generation and validation. The requirements engineer models the requirements in the requirements document by assigning relations between them with tool support in [12]. The software architect designs the software architecture for the requirements and can assign some initial traces between requirements and architecture.

The modeling process is forked into three activities: reformulating requirements, generating trace and validating trace.

**Reformulating Requirements:** This activity takes the input requirements model and input architectural model as input and produces the reformulated requirement as output. The software
architect reformulates the requirements in terms of logical formulas over the architecture.

**Verifying Architecture:** This activity takes the input architectural model and the reformulated requirement as input and produces execution trace or counter example (see Section 3.1). The activity checks whether the requirements are satisfied by the architecture. The activity is done in Maude automatically.

**Generating Trace:** This activity takes the input trace, requirements and architecture models with the output of verifying architecture as input and produces output trace and requirements models as output. If the activity uses only requirements relations in the requirements model and initial traces in the input trace model to generate traces and/or requirements relations, it is sufficient to perform this activity after modeling requirements & designing architecture activity only. The activity is automatic.

**Validating Trace:** This activity takes the input trace model, input requirements model, input architecture model as input and produces an output error model as output. This activity is automatic. However, the interpretation of the output of this activity (the output error model) with the trace model should be done by the software architect manually.

**Iterating:** The process given in Figure 2 is iterative. The requirements engineer and/or the software architect may return to the requirements modeling & designing architecture activity in order to fix requirements relations and/or traces in the output models. If there is no need to update the models, the process is terminated.

In order to implement the tool, we successively provide the followings:

- **Trace Metamodel.** We use a trace metamodel to provide a structure to traces. The metamodel includes most commonly found entities in trace metamodel in literature, and requirements & architecture specific traces. We use two types of traces between requirements and architecture: AllocatedTo and Satisfies.

- **Semantics of Traces.** In the literature, traces in the trace metamodel are informally defined. We formalize requirements, architecture and traces between them by using FOL.

The tool uses traces with requirements relations. Therefore, we need semantics of requirements relations.

- **Semantics of Requirements Relations.** We identified five types of requirements relations: requires, refines, partially refines, contains, and conflicts. The requirements metamodel together with semantics of requirements relations in FOL are already given in [12].

- **Architecture and Verification.** The software architecture is expressed in Architecture Design and Analysis Language (AADL) [2]. We have defined dynamic semantics for part of AADL in terms of rewriting logic supported by the Maude language and tools [6] [7]. This enables performing simulation and verification of AADL models [29]. For the verification, architectural significant functional requirements are reformulated as formalized scenarios and then linear temporal properties are checked using linear temporal logic (LTL) [3]. Applying verification techniques for requirements is not the main focus of our paper. The details can be found in [28].

- **Generating and Validating Traces.** We use semantics of traces and requirements relations with architecture verification techniques for generating and validating traces.

### 3. FEATURES OF THE TOOL

#### 3.1 Verification of Architecture for Functional Requirements

We limit ourselves to verification of functional requirements only. The purpose of the verification is to check if requirements are correctly implemented in the architecture. The tool uses verification results in both trace generation and trace validation as an input. Figure 3 illustrates the verification of architecture for functional requirements.

![Figure 3 Verification of Architecture for Functional Requirements](image)

The verification is represented by the Satisfies and ConformsTo relations in Figure 3. ConformsTo implies that the state space captures the specified properties. We have the following artifacts in the verification of architecture:

- **Functional Requirements.** Requirements which describe the functions that the system is to execute; for example, formatting some text or receiving a data.

- **Architecture in AADL.** The architecture of the system to be built. It plays the role of the solution for the problem defined by the requirements.

- **Property Specifications in Maude.** The formal description of the required behavior of the architecture. The requirements are reformulated as properties in terms of the solution, which is the architecture (reformulate and uses in Figure 3). These properties are checked for the architecture by the model checker. The requirement is first described as a formalized scenario, and then described as property specification [28]. The property specification uses any logic such as Linear Temporal Logic (LTL), First-Order Logic (FOL), or Computation-Free Logic (CTL). In the tool, we use the formal analysis features of Maude.

- **State Space in Maude.** The presence of a dynamic semantics specification of AADL in Maude makes the architectural models executable. The architecture is executed and a state space is produced (simulate in Figure 3). This execution simulates the behavior of the system on the architecture level so that it can be studied to see how the system will work. Discrete event simulation, which introduces the notion of events, states, and state space, is used. A state describes the loci of data values within the architecture. Two states are connected by a transition and all states are captured by the state space. The result of the verification, which might be a counter example or execution trace, is used to generate and validate traces. An execution trace is the ordered set of states which are generated where the reformulated requirement is satisfied. A counter example is the ordered set of states which are
generated where the reformulated requirement is not satisfied. Since the focus of this paper is not verification and simulation, we do not give details of the AADL semantics used in the tool. This is itself a non-trivial topic and subject of another work.

### 3.2 Generation of Traces

Generating traces aims at deducing traces between requirements and architecture based solely on verification of architecture and/or the requirements relations in the requirements model. Our tool does not need initial traces to generate new traces. The tool uses the result of the verification of architecture. If the verification is successful, the architecture satisfies the requirement. According to the semantics of traces in [13], the Satisfies trace is generated between the architectural elements in the execution trace and the requirement. These elements collectively satisfy the requirement and form the part of the architecture to which the requirement is traced. A counter example means that although the requirement is allocated to the architectural elements, the architecture does not satisfy it. The AllocatedTo trace can be generated but the Satisfies does not hold.

The second way to generate traces is to use the requirements relations. The constraints about traces are derived from the semantics of traces and the semantics of requirements relations (see [13]). The constraints ensure that requirements relations are preserved in the architecture by the satisfying elements. Our tool uses the constraints also to generate requirements relations from traces.

The verification result, and therefore the traces, depends on the reformulation of the requirement to be checked. The software architect needs to consider potential false positive and missed traces. Such traces are defined in relation to the set of actual traces, which is the golden standard for a pair of requirements and architecture.

### 3.3 Validation of Traces

Validation aims at identifying the traces which do not obey the trace semantics. Our tool uses the semantics of requirements relations together with the trace semantics to validate traces which are already generated or assigned by the architect. Checking is performed according to the constraints derived from the semantics of traces and requirements relations. The following is an example constraint to be checked in the trace model by the tool:

\[ E_{A1} \supseteq E_{A2} \text{ if } (R_1 \text{ Contains } R_2) \land (E_{A1} \text{ Satisfies } R_1) \land (E_{A2} \text{ Satisfies } R_2) \]

where \( E_{A1} \) and \( E_{A2} \) are sets of architectural elements, and \( R_1 \) and \( R_2 \) are requirements. The tool identifies traces or requirements relations which violate the constraints. Validation using requirements relations can be used in two ways. First, the software architect may decide that an invalid trace is a true positive and then he reconsiders the requirements relations. Second, the software architect decides that requirements relations are all valid, then, he identifies the invalid traces.

Our tool also provides validation of traces by using verification results. The architect assigns some AllocatedTo traces while creating the architecture. In order to ensure that the architecture satisfies the requirements, the verification of architecture is processed. For the requirements satisfied by the architecture, the Satisfies traces are generated by the tool. The assigned AllocatedTo traces and the generated Satisfies traces are compared by the tool. These traces are validated based on the comparison.

The tool finds the differences and intersections of the sets of the traces. The software architect should check especially the difference of the sets and decide about the validity of traces. For the requirements which are not satisfied by the architecture, the AllocatedTo traces are generated from the counter example by the tool. The assigned and generated AllocatedTo traces are validated based on the comparison of trace sets.

The requirement may describe a complex system property amenable to decomposition. The tool can not trace to the part of the requirement responsible for a failure. The requirements engineer should decompose the requirement into sub-parts (Contains relation) until each requirement describes only one property.

### 4. ARCHITECTURE OF THE TOOL

The tool contains five components (rounded boxes in Figure 4).

**Figure 4 Architecture of the Tool**

**Architecture Verification in Maude:** The input for architecture verification component is the input architecture model and the requirement(s) reformulated as LTL. This component is used in the trace generation. The verification and simulation are performed by the model checker and the rule execution engine of Maude. The architectural model originally expressed in AADL is transformed to Maude terms. The AADL metamodel is encoded as a set of sorts. The dynamic semantics of AADL is given in rewriting rules. Requirements are reformulated as LTL formulas, the language supported by Maude checker.

**Trace Generator using Verification Result in ATL:** The input of the component is the execution trace and counter example. The component is implemented as an ATL transformation. If the verification result is an execution trace, the Satisfies traces are generated between the checked requirement(s) and the architectural elements in the execution trace. If the verification result is a counter example, the AllocatedTo traces are generated between the checked requirement(s) and the architectural elements marked in the counter example. The output is given in the output trace model 1.

**Trace Generator using Requirements Relations in ATL:** The input of the component is the input architecture model, input trace model, and input requirements model. The component is used in the trace generation. It is implemented as an ATL model transformation. The component generates new traces based on requirements relations in the input requirements model and the
constraints in Figure 1. The output is given in the output trace model.

For output of the two trace generator components, we use two different output trace models in order to state that the outputs do not have to be the same. In the generation part of Figure 1 which is generating traces by using requirements relations and verification of architecture, the three components above are used. First, the traces are generated in the output trace model 1 by the component trace generator by using verification result. Then the output trace model 1 is used as an input trace model by the component trace generator by using requirements relations to generate traces based on requirements relations in the input requirements model.

**Trace Validator in ATL:** The input of the component is the input architecture model, input trace model, and input requirements model. The component is used in the trace validation part of all scenarios. It is implemented as an ATL transformation. The component checks the validity of assigned traces between R&A by using verification output or requirements relations. It can also check the validity of requirements relations by using traces between R&A. The output is the output error model which contains invalid traces and invalid requirements relations.

**Requirements Relation Generator using Traces in ATL:** The input of the component is the input architecture model, input trace model, and input requirements model. The component is used in the trace generation part of Figure 1. It is implemented as an ATL model transformation. The output is given in the output requirements model which contains only the generated requirements relations.

5. EVALUATION OF THE TOOL

Our tool can be evaluated from different perspectives like usability, performance and scalability. The usability of the tool mainly relies on the Eclipse environment. For the output of the counterexample and execution traces, no GUI is provided. For the prototype we consider this to be acceptable. In this section, we conduct performance and scalability tests of the tool for generating and validating traces. Our tool uses model checking techniques in verification of architecture for functional requirements. Since these techniques may have problems in handling large amounts of model elements and states, the performance and scalability of our tool mainly depends on the scalability of the model checking algorithms in Maude. Therefore, we focus on the model checking part of our tool in our performance and scalability tests.

**Performance testing** is conducted to evaluate the compliance of a system or component with specified performance requirements [31]. The requirement in our test is that the tool performs in reasonable time (say less than one minute) with average number of architecture elements. We base our estimate for the average number of architectural elements on a report by MacCormack et al. [22]. They characterize the differences in design structure between complex software products like Mozilla and Linux. The report shows that the architectural model of a real system contains around 2000 model elements. We take this finding as a base for our performance tests.

**Scalability testing** is a performance testing focused on ensuring the application under test gracefully handles increases in workload [31]. The workload in our performance test is the number of states. Our interpretation of scalability for evaluating the tool is the following: the tool scales if the time spent by the tool increases linearly when the number of generated states increases linearly.

Our dependent variable in the performance and scalability tests is the time for simulation and verification (in seconds). The independent variable used in the performance tests is number of elements in the architecture. We define the number of elements as follows: number of component instances + number of feature instances + number of port connections where component, feature and port connections are the architectural elements in AADL. The independent variable used in the scalability test is the number of states generated in the simulation. We define the number of states in the simulation as follows: the number of states the simulation is enforced to explore. These two variables are closely related to each other. If the number of elements is increased, it is likely that the number of states required for simulation and verification also increases. However, this does not always have to be the case. For example, if we introduce a new system property to the architecture, not related to the existing system properties, we do not have to increase the number of states in the simulation and verification of architecture for existing system properties.

Memory consumption is not measured in the performance tests. The runs for each performance test are executed six times. The average for each run is derived from six executions. The performance tests are done with Intel Core 2 Duo i5 running at 2.67 GHz, and 2.99GB of memory, running Kubuntu 9.10. We use Core Maude 2.4 for Linux. The models used in the performance tests are artificially created to test the tool with certain number of elements and states.

**Performance test.** The test is set up as follows. We increase the number of elements by adding components, data ports and connections to the architecture. We start with 2000 architectural elements and end up with 3000 architectural elements. The number of states for each run is 500, 1000 and 2000. The results of the performance test are shown in Table 1. Since the results of the performance test might be different when the verification result is an execution trace or a counter example, the performance test is done for both cases (see Table 1(a) and Table 1(b)). The standard deviation of the data is approximately 0.3%.

<table>
<thead>
<tr>
<th>Simulation Time (sec) for the Execution Trace</th>
<th># elements</th>
<th># states = 500</th>
<th># states = 1000</th>
<th># states = 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>7.8</td>
<td>15.9</td>
<td>33.8</td>
<td></td>
</tr>
<tr>
<td>2200</td>
<td>8.7</td>
<td>17.5</td>
<td>37.2</td>
<td></td>
</tr>
<tr>
<td>2400</td>
<td>9.3</td>
<td>19.4</td>
<td>40.4</td>
<td></td>
</tr>
<tr>
<td>2600</td>
<td>10.1</td>
<td>20.9</td>
<td>43.3</td>
<td></td>
</tr>
<tr>
<td>2800</td>
<td>10.9</td>
<td>22.4</td>
<td>46.5</td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td>11.5</td>
<td>23.9</td>
<td>49.6</td>
<td></td>
</tr>
</tbody>
</table>

(a) Simulation with Execution Trace

<table>
<thead>
<tr>
<th>Simulation Time (sec) for the Counter Example</th>
<th># elements</th>
<th># states = 500</th>
<th># states = 1000</th>
<th># states = 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>2.6</td>
<td>5.2</td>
<td>10.8</td>
<td></td>
</tr>
<tr>
<td>2200</td>
<td>2.8</td>
<td>5.7</td>
<td>11.9</td>
<td></td>
</tr>
<tr>
<td>2400</td>
<td>3.1</td>
<td>6.3</td>
<td>13.0</td>
<td></td>
</tr>
<tr>
<td>2600</td>
<td>3.3</td>
<td>6.7</td>
<td>14.0</td>
<td></td>
</tr>
<tr>
<td>2800</td>
<td>3.5</td>
<td>7.2</td>
<td>15.2</td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td>3.7</td>
<td>7.7</td>
<td>16.1</td>
<td></td>
</tr>
</tbody>
</table>
According to these performance tests, the tool performs below one minute with average number of architecture elements in a real system. The increase in the simulation time is linear and up to 50 seconds for 2000 states (see Figure 5).

Scalability test. The goal of this test is to investigate how the tool handles increases in the number of states over several orders of magnitude. Our independent variable is the number of states. We also compare the scalability test results of the tool using Maude with the results of the tool using different simulation and verification environments such as Alloy [17]. Therefore, same execution semantics of AADL in Maude are encoded in Alloy. The first part of the performance test is done in Maude with 10000 architectural elements (see Table 2(a)). Then, the second part of the performance test is done in Alloy (see Table 2(b)). In [20], we investigated simulation and verification in Alloy. Based on our experience, we already know that Alloy is not suitable for large amounts of model elements and states. Therefore, we choose to run the second part of the performance test in Alloy with small number of architecture elements (38 elements) (see Table 2(b)).

According to the scalability test results of our tool using Maude, the simulation time increases linearly when the number of states increases linearly (see Figure 6). We ran out of memory in Maude when we try simulation for 10000 architectural elements with 5000 states. For Alloy, the simulation time also increases linearly when the number of states increases, however, for much smaller number of architectural elements and much smaller number of states.

According to these test results, we conclude that our tool scales much better over a broad range of states for more realistic architectures sizes with Maude than with Alloy.

We cover a subset of AADL semantics in our tool. Our results are valid for this subset. The performance test results may change with more AADL semantics encoded as state transitions in Maude. There is another tool [23] [27] for the representation of AADL models in Maude which covers more execution semantics of AADL. However, we needed our own state transitions for trace generation and validation purposes. As a future work, we plan to integrate the tool in [23] [27] with our tool.

6. RELATED WORK
A number of approaches with tool support describe generating and validating traces. Egyed et al. [9] [10] provides an automated traceability approach that uses a small number of traces as input. Similarly to this work, we use reformulation of requirements as scenarios. In [9] [10], the source code is executed according to the scenarios and then traces are generated between requirements and source code. Footprint graph is used to detect the incomplete and incorrect input to the approach. Dependencies between requirements can be detected based on overlaps among the lines of code implementing the requirements. However, there is only one type of traces and

---

**Figure 5 Simulation Time as a Function of the Number of Architectural Elements**

According to these performance tests, the tool performs below one minute with average number of architecture elements in a real system. The increase in the simulation time is linear and up to 50 seconds for 2000 states (see Figure 5).

<table>
<thead>
<tr>
<th>Number of States</th>
<th>Simulation Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1.5</td>
</tr>
<tr>
<td>100</td>
<td>9.5</td>
</tr>
<tr>
<td>1000</td>
<td>82.1</td>
</tr>
<tr>
<td>3000</td>
<td>285.4</td>
</tr>
<tr>
<td>4500</td>
<td>401.8</td>
</tr>
</tbody>
</table>

**Table 2 Simulation Times in the Scalability Test**

<table>
<thead>
<tr>
<th>Number of States</th>
<th>Simulation Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>14.2</td>
</tr>
<tr>
<td>40</td>
<td>53.7</td>
</tr>
<tr>
<td>60</td>
<td>105.8</td>
</tr>
<tr>
<td>80</td>
<td>180.4</td>
</tr>
<tr>
<td>100</td>
<td>300.9</td>
</tr>
</tbody>
</table>

---

**Figure 6 Simulation Time vs. Number of States in Alloy and Maude**

According to the scalability test results of our tool using Maude, the simulation time increases linearly when the number of states increases linearly (see Figure 6). We ran out of memory in Maude when we try simulation for 10000 architectural elements with 5000 states. For Alloy, the simulation time also increases linearly when the number of states increases, however, for much smaller number of architectural elements and much smaller number of states.

According to these test results, we conclude that our tool scales much better over a broad range of states for more realistic architectures sizes with Maude than with Alloy.
requirements relations in [9] [10]. In our tool, we have multiple trace and requirements relation types.

The System Modeling Language (SysML) [26] is a domain specific modeling language for system engineering. It provides modeling constructs to represent text-based requirements and relate them to other modeling elements such as architectural elements with stereotypes. Types of traces between requirements and architecture are given with informal textual definitions in SysML. However, the SysML standard does not provide any formal definitions of trace types.

Schwarz et al. [30] describe a graph-based traceability approach with tool support. Generation and maintenance of traces are handled by model transformations. The \textit{Satisfies} trace is provided without any formal semantics or textual definition. Components, interfaces and ports in the architecture are created automatically from requirements and use cases by using heuristics. \textit{Satisfies} traces are generated in result. In our approach, architecture is created manually and then the traces are generated and validated.

The tool by Grechanik et al. [15] supports generating traces between types and variables in Java programs and elements of use-case diagrams (UCD). The tool combines program analysis, run-time monitoring, and machine learning to generate traces. Initial traces are needed for trace generation. Relations between program entities are compared with corresponding relations between elements in UCDs only to validate traces.

Mader et al. [21] focus on maintaining traces between requirements and UML models for the changes in UML models. Patterns for maintaining traces are specified based on the classification of UML model changes. The tool in [21] is complementary to our tool.


Egyed [11] introduces the UML/Analyzer tool which does consistency checking based on model transformation. Abi-Antoun and Medvidovic [1] describes a semi-automatic approach to assist in refining a high-level architecture specified in an architecture description language into a design described with UML. The works in [1] and [11] are complementary in the sense that one describes the refinement and the other one is providing how to ensure the preservation of properties in this refinement. Heckel and Thone [16] propose a notion of refinement which requires the preservation of both structural and behavior at the lower level of abstraction. Based on formal refinement relationships between abstract and more concrete architectural models, they use model checking techniques to verify that abstract scenarios can also be realized in the more concrete architecture. Most of the works given above focuses on the generation and validation of traces between architectural and detailed design models.

7. CONCLUSIONS

In this paper, we presented a tool that provides trace establishment by using semantics of traces between R&A (Requirements and Architecture). The tool uses Maude, a formal language based on equational and rewriting logic, and MDE technologies such as Eclipse EMF and ATL.

There are some open issues in the usage of the tool. Reformulation of requirements in terms of solution domain is a part of design process and is hard to automate. The architect might still need to check the generated traces. In case of false positives the requirements model and relations should be checked. This suggests an iterative semiautomatic process of using our tool. In such a process, the software architect can gradually improve the quality of the traces and the requirements. Case studies conducted with the industry [5] shows that LTL/CTL is hard to reformulate and check requirements in the architecture. Domain-specific languages can be used for requirements of certain type that allow compilation of LTL/CTL formulas [5]. Starting from natural language text, Semantics Business Vocabulary and Rules (SBVR) [25] can support reformulation requirements in terms of LTL/CTL formulas. Extending our tool with this kind of languages will ease the reformulation of requirements.

Maintenance of traces is not covered in this paper. We focus on generating and validating traces as a first step. In case of changes in the requirements or in the architecture, some of the traces will be invalid and incomplete. Our tool needs to be extended for maintenance of traces.

We mainly focused on scalability issues in our tool for generating and validating traces. Since model checking techniques may have problems with big size of models and number of states, the scalability of our tool depends on the scalability of the model checking algorithms in Maude. Our tool needs further improvement for usability. The core parts of the tool are implemented. However, integration of these parts is currently done manually and we need a user interface to control all these parts.

8. REFERENCES


[23] Moment2-AADL.
http://www.cs.usc.edu/people/aboronat/tools/moment2-aadl/


An Aspect-based Traceability Mechanism for Domain Specific Languages

Rolf-Helge Pfeiffer
IT University of Copenhagen, Denmark
ropf@itu.dk

Andrzej Wąsowski
IT University of Copenhagen, Denmark
wasowski@itu.dk

ABSTRACT
Development environments for domain specific modeling usually represent elements of visual models as objects when in memory and as XML elements when persisted. Visual models are editable using different kinds of editors, and both the in-memory representations and the serialization syntax can be manipulated by automatic tools. We present Tengja, a toolkit, that automatically collects the traces between model elements in abstract, visual, and serialization syntax. Once the trace model is established by Tengja it can be used by other applications to synchronize representations involved, or to navigate across models. We demonstrate the toolkit by implementing a simple navigation support on top of it.

Keywords
Model-Driven Software Development, Traceability, Aspect-Oriented Software Development

1. INTRODUCTION

Modeling languages can be classified into visual and textual ones. Textual languages, such as XML, use alphabet characters to represent models. The Unified Modeling Language (UML) is an example of a visual modeling language. The language specification, gives its syntax using visual elements [1]. Visual and textual modeling languages are used on different levels of abstraction. Often, the more abstract a model is, the more likely a visual notation is used.

Somewhat controversially, we claim that visual models do not exist in practice. Instead textual models are interpreted by tools and represented visually. The visualizations vary, depending on the editors used; ranging from text processors, via structured forms and trees, to fully fledged diagramming editors. Often several views are used for the same model.

State of the art frameworks, such as Eclipse Modeling Framework (EMF) and Graphical Modeling Framework (GMF), allow automatic generation of rich editors for domain specific languages (DSLs). The editors are used by developers to produce new software artifacts—for instance specifications of product variants in model-driven product line architectures [16]. Since the editors rely solely on a description of abstract syntax supplemented with visual diagram meta-data, a large degree of code reuse can be achieved.

Inclusion in mature development processes, imposes high requirements on modeling editors. Not only should they support developers in the same manner as usual programming environments, but also provide modeling-specific functionalities. These, among others, require supporting concurrent editing of multiple views of an interconnected network of models by multiple developers using different physical workstations. Since such applications often need to rely on serialized versions of models, we approach the question of traceability of model elements to their serialized representations.

A modern DSL editor distinguishes three representations:

- The serialization concrete syntax (serialization syntax for short) is the persistent representation of models. It often takes form of an XML file adhering to a particular schema. Tools use serialization syntax at least for storage, but also for transformation, versioning, etc.

- The abstract syntax is the object graph representing the domain specific model in memory. It takes the form of an object model adhering to a particular class model (the meta-model). The abstract syntax is what most researchers consider ‘a model’.

- The visual concrete syntax (or visual syntax for short) is the diagram shown to the user in a visual editor. The visual syntax is how the users perceive models.

In programming languages the visual and the serialization syntax coincide: developers work with the textual representation, which is directly stored in files. For domain specific modeling languages this is rarely so: The Eclipse DSL toolkit [9], Generic Modeling Environment [6], MetaEdit+ [10] and Microsoft DSL Tools [7], all hide the serialization syntax from users. In all these tools editors work with visual syntax, while abstract syntax and serialization syntax are used for transformation, synchronization and storage.

Modeling frameworks uniformly support loading and persisting models—a form of model-to-model traceability between the serialization and the abstract syntax. For the visual syntax there are well developed traceability mechanisms linking it to the abstract syntax: all frameworks use them to realize the Model-View-Controller (MVC) pattern [15], where changes to the visual syntax are immediately synchronized with the abstract syntax. This is a form of element-to-element traceability between the abstract and visual models.
However, presently, element-to-element traceability between the abstract and the serialization syntax is not supported. The ability to link abstract and serialization syntax at the element granularity enables the following use cases:

- **Model debugging** — a developer can navigate from visual syntax elements directly to XMI representation, in order to inspect the values saved by the editor. This is useful to debug various phases of the editor, to debug the models, and to debug model transformations. This use case is supported by our prototype (below).

- **Monitoring and navigating cross-model soft references** — stating soft references across models at the level of serialization syntax allows using external XML processing tools and XML transformations to manage models, while still maintaining inter-model consistency. Traceability between abstract syntax elements and the serialization syntax allows monitoring external changes to the abstract syntax, or to the concrete syntax, for example in order to notify the user about locks to a part of the model, or about concurrent updates.

- **Interactive creation of language independent soft references** — extending the GUI of interactive editors to support definition of soft references and ‘anchors’ for higher-level trace-based tools. A user could indicate a model element, and annotate it with a dependency constraint on another system component. This dependency can be stored as a dependency between the serialized versions of the element and that component, which can be processed by a traceability tool that is oblivious to the specific notion of models and components, but just knows about the concept of anchor and dependency. We are presently working on building such a tool for tracing between heterogenous artifacts.

In this paper we present **Tengja**\(^1\), a toolkit for element-to-element tracing from visual and abstract syntax to their serialized versions. Tengja is implemented as an aspect in AspectJ, which automatically establishes element-to-element links, by analyzing the process of saving the model. Our toolkit is non-invasive and highly reusable due to its aspect oriented nature. It works for existing models in EMF and GMF. Neither the models nor the editors need to be modified. It works with all Eclipse model editors (including tree editors, and GMF generated domain specific editors), as long as they rely on the standard persistence mechanism.

\(^1\)Tengja, Icelandic for *connect*, was chosen to avoid conflicts with ‘connects’, ‘connections’, and ‘connectors’ appearing frequently in model-driven development literature.
As a proof of concept, which demonstrates the effectiveness of the framework, we develop a simple debugging facility using the Tengja-toolkit. It extends the current Eclipse editors, by introducing an ability to highlight an element and request its view in other representations. For example, in Figure 1 the user highlights EAttribute element of the model opened in a diagram editor of GMF (left top). On user request, the framework automatically opens three other editors highlighting the corresponding model parts in each of them (a tree view to the right, an Ecore XML model to bottom-right and an Ecore diagram model serialized to XML to bottom-left). The user can then proceed to debug the modeling editors, the serialization mechanism, or the model itself, by inspecting or modifying the representations shown. As mentioned above the same mechanism could be used to trigger warnings, errors or updates depending on automatic or concurrent changes to the serialized or abstract syntax. We intend to investigate such applications in future.

We proceed as follows. Section 2 provides background on modeling languages and visualizations in Eclipse. In Section 3 we present Tengja itself, and evaluate it in Section 4. Sections 5–7 discuss our solution, compare it to published sources, indicate future research directions and conclude.

2. MODELS IN ECLIPSE

In this paper we work with EMF and GMF—both visual modeling components of the Eclipse DSL toolkit. Both frameworks are representative of modern environments for model-driven software development. Eclipse, GMF and EMF together enable developers to easily define their DSLs and to generate specialized editors for them.

Eclipse provides three different kinds of model editors: diagram editors (DiagramDocumentEditor as part of GMF), structured tree editors (Tree used by EMF), and text editors. Diagram editors and tree editors allow interacting with visual syntax. Text editors allow for editing models in serialization syntax or in other textual representations. Eclipse’s modeling package contains examples of a few predefined editors, such like the UML class diagram editor. In Figure 1 the left-topmost model is shown in a diagram editor (in fact in the class diagram editor), the right-topmost one in a tree editor, while the two others are shown in usual text editors.

Presently, the XML Metadata Interchange (XMI) format is used to persist models in Eclipse. The following gives an overview over the artifacts that are used to store the relevant information of a graphical model using the example of the Ecore Metamodeling Language [2].

The model in the top-right of Figure 1 is an excerpt of the EcoreOverview.ecore data model presented in a tree editor. Its serialization to an XMI file is shown underneath (bottom-right). Notice, that since the tree editor contains no model specific layout information, the serialization only contains data model elements. To the left (top-left) a visual diagram of the same model is shown using UML-like class diagram syntax. In physical world this image of boxes and lines is the actual model as it would appear on paper. However from the tool perspective this is just one possible visualization of the model. Finally, the bottom-left of the figure shows the serialization of the layout information of the class diagram.

Eclipse separates the visual information model from the actual data model, spreading their persistent representations over two files. These are integrated together by modeling editors following the MVC pattern [15]. Figure 2 illustrates the steps performed when loading, editing, and saving the EcoreOverview.ecorediag model. When an editor is opened, the visual information model is loaded first, then the data model is loaded, and both are interpreted, before the model is presented to the user (the left part of Figure 2).

3. TENGJA

Our aim is to provide an extension to Eclipse, which recovers the links between the serialization syntax and the abstract syntax of models. Since in general, visual modeling languages are just graphs with different node types and different edge types, frameworks deploy graph traversal algorithms to persist the models.

The gear-wheels in Figure 3(a) symbolize the standard persistence mechanism of Eclipse serializing both the visual diagram information and the data model one after another. Both GMF and EMF use the mechanism implemented in the org.eclipse.emf.ecore.emf.ecoreImpl package for this. The XMI representation is generated by class XMLSaveImpl. It traverses the in-memory object graph and feature-wise generates the corresponding XML elements. If there exist corresponding, less abstract models, for instance corresponding data models, the save mechanism is called iteratively on all these dependent models.

Initially, we had anticipated to find a compositional bottom-up graph traversal algorithm that generates a block of serialization syntax for each model element. In reality the translation of each model element is scattered over multiple methods that are called sequentially. The model graph is not traversed bottom-up from simple element leaves to the top root. Rather, the persistent model is constructed sequentially starting from the root model element. The process does not provide any context anchor that would capture information about the scope: which model element is in scope and what is the serialization syntax generated for it. As a result, first it is challenging to understand the relations of a particular model element and its persistent representation, and, second, no explicit traces between a graphical model element and its serialization syntax are kept.

Since the standard persistence mechanism obscures the traces, and since we aim at a reusable and non-invasive trace...
4. EXPERIMENTAL EVALUATION

We evaluate robustness of Tengja with a semi-automated test. A test program loads a number of models and requests traceability links for their elements. The results are stored in a log, which is verified by a human expert.

We used a number of GMF diagram models and a number of EMF data models to run the evaluation. The models where chosen to represent different sizes and use in real projects. In Tables 1 and 2 column headers correspond to individual models. Ecore.ecore, EcoreAnnotations.ecorediag, EcoreDataTypes.ecorediag, EcoreGenerics.ecorediag, EcoreHierarchy.ecorediag, EcoreOverview.ecorediag, XMLNamespace.ecore, and XMLType.ecore are part of the Ecore implementation available in the org.eclipse.emf.ecore Eclipse plug-in (we used version 2.5.0.v200906151043). The model fmp.ecore is the metamodel of feature models as used in the Feature Modeling Plug-in [5], available online.

The robustness evaluation was conducted by running a test program on the above models. The program automatically opens the models in their corresponding editors and generates the mapping between model elements and their textual representations by saving the respective model before opening it. Subsequently, the test iterates over all elements of each model and stores their object identifiers with the textual representation to a log file. The generated log files were manually compared against the source files of the corresponding model. The results of this check, sorted by the type of model elements, can be found in Table 1 and in Table 2. In each table cell x/y means that x model elements out of y present in the model were correctly traced, whereas “n/a” means that the corresponding model does not contain a model element of the given type.

The result of the evaluation was positive, showing 93% recall and 100% precision. This means that 93% of the elements have been traced, and all of them traced correctly.

The only exception are model elements of type EGenericType (Table 1). Tengja is able to map graphical model objects to their serialization syntax only if the textual representation is a complete XML element: a string of either “<.../>” or “<identifier>...</identifier>” form. Most often, model elements of type EGenericType are in neither form, instead they are subparts of an XML element. We do not consider this a serious problem, since elements of this type are not ‘clickable’ in model editors anyway. If we only consider clickable elements the recall raises to 100% with the same precision.

Threats to Validity. There are two main threats to validity of this experiment. First, our assessment of logs of traces on evaluation models have been performed by the implementer of Tengja, which could introduce a bias. Moreover
the evaluation targets were all simple class-diagram files, not actual domain specific models. We intend to expand the set of evaluation models, and improve the independence of the evaluation in our ongoing work on this project.

5. DISCUSSION

Tengja is highly reusable, in that it minimizes the amount of code that need to be refactored when adapting it to a new modeling framework—only the pointcut specifications should be refactored. In this sense Tengja is ready to support modeling frameworks of Eclipse, that do not exist as of today. Tengja is non-invasive meaning that it does not require any modifications to models, metamodels, or existing editors. We have used AspectJ to reach these objectives, but we do believe that using other composition mechanisms, such as Object Teams, would yield similar results.

Alternatively we could have implemented an invasive solution, which completely replaces the persistence mechanism for one, which generates the trace model on the fly. However this requires deep changes in Eclipse’s implementation and would not be reusable across new modeling frameworks.

In model transformation systems with QVT-like architecture the trace models are automatically created while transformations are applied. Implementing the saving mechanism in such a system, would gain traceability for free, albeit still just for one particular framework at a time. Presently QVT itself does not standardize transformations involving serialization syntax, but we would need a language that supports both automatic element-to-element traceability storage and model-to-text transformations. One way to obtain this would be to weave an aspect similar to Tengja into an implementation of existing transformation toolkit.

One could disregard working with serialization syntax at all. Indeed much of the functionality can be achieved at the model level, using soft references and alike. However we have to recognize that a file is the single most popular unit of organizing software development artifacts. Interfaces for software development artifacts tend to be the simplest to implement based on files and they allow programmers to access them with regular text editors (which is very popular).

Since Tengja extends the standard persistence mechanism and supports any Ecore-based DSLs, it is directly interop-
erable not only with editors, but also with other modeling technologies of Eclipse, including transformation languages like XTend. For example, it can recover traces from abstract to serialization syntax for models resulting out of transformations. If the transformation framework supports materialization of traces, these could be combined with our traces, in order to provide complete end-to-end traceability for chains of transformations.

Tenga assumes a rather close relation between the abstract and serialization syntax. We believe that this is not a serious limitation—it has been expressive enough to succeed on a handful of DSLs of the Eclipse project, which we have used for initial evaluation. Indeed, we hypothesize that, unlike for visual syntax [14], the relation between the abstract syntax and concrete serialization syntax tends to be close for most modeling languages, as it is also known in compilers and interpreters for programming languages. Normally, there exists a mapping between the abstract and serialized representations, where elements of the abstract syntax map to convex fragments of serialization syntax. This assumption significantly simplifies our implementation.

6. RELATED WORK

Traceability between abstract and serialization syntax is hardly discussed in modeling community. Oldevik and Neple [13] discuss it in the context of OMG’s model-to-text transformation standard. They propose to automatically generate trace models to textual models while the transformation applies, essentially in the same manner as it is done for model-to-model transformations. Our work recovers these links automatically without directly modifying the persistence code, which would otherwise be required to realize their vision. In this sense, Tenga is a kind of model transformation [8, 12], which recovers a trace model by instrumenting a model-to-text transformation.

The literature is abundant in meta-models of trace languages, and in frameworks allowing defining, maintaining and querying traceability information. Much less attention is devoted to automatic recovery of such links. One example is the work of Antkiewicz and coauthors [3] on recovery of framework specific models from Java code. They use static analysis to recover models from Java source code (arguably a textual representation). Once the models are extracted, trace models are used to maintain synchronization links [4].

Ráth et al. [14] extend trace models to arbitrary relations between the abstract and the visual syntax. They use incremental model transformations to maintain the two layers in sync. While such a generality of constraints appears superfluous between the serialization and abstract syntax, it would still be interesting to see whether their technology could keep textual and visual models synchronized easily.

In [11] a rule based approach is proposed for automatic updating of traceability rules based on common model editing operations. It would be interesting to see whether this adapts also to automatic model transformations, and to links between serialization and abstract syntax, in particular in scenarios when the concrete representation is changed concurrently to the abstract representation.

7. CONCLUSION AND FUTURE WORK

We have presented Tenga, a robust, non-invasive and reusable aspect-oriented solution for automatic harvesting of traceability links between abstract and serialization syntaxes in modeling frameworks of Eclipse. We are not aware of any tool with similar objectives being available so far.

Tenga visualizes traces between elements of interrelated models on multiple levels of abstraction. Thereby, it may enhance the awareness and the understanding of traces. To demonstrate this we have developed a simple extension for navigation between model elements in various representations, which can be used to debug models and editors.

In future we will continue developing Tenga. For example, the tool presently does not handle models with multiple root elements. This is going to be addressed. We intend to use Tenga to develop higher level tools, for example supporting compositional reasoning about systems in the style of component algebra. Tenga will be the main linking mechanism of this prospective tool.

8. REFERENCES


ABSTRACT
Although traceability is often a suggested requirement for general software development, there are areas such as airborne systems, where traceability is a compulsory part of the development process. This paper describes a tool chain that is able to generate and to follow traceability links across model-to-model and model-to-code transformations, and capable of providing navigability support along these traceability links. We elaborate on the conceptual design of our tool chain and provide details on its realization in a DSML environment underpinned by graph rewriting-based model transformation.

Categories and Subject Descriptors
H.4 [Information Systems Applications]: Miscellaneous; D.2.8 [Software Engineering]: Metrics—complexity measures, performance measures

General Terms
Theory

Keywords
Traceability, Model transformation

1. INTRODUCTION
Traceability is one of the most crucial software issues in airborne systems. The document describing the software considerations for airborne systems [13] provides the following traceability guidance. (i) Traceability between the system requirements and software requirements is required. This enables the verification of the implementation of a concrete requirement and the derived requirements. Moreover, it allows one to verify the completeness of the implemented requirements. (ii) The traceability between the high-level requirements and low-level requirements is required. Apart from the completeness, this gives a visibility to the derived requirements. (iii) Traceability between source code and low-level requirements is required. This enables the discovery of un-related or un-documented source code in the complete implementation of the low-level requirements. During the verification process these traceability links between the artifacts must be examined.

Figure 1: Main concepts

Model-Driven Development (MDD) is a popular and significant paradigm within software engineering. Underpinned by metamodeling techniques, Domain-Specific Modeling Languages (DSMLs) [17] offer an excellent conceptual and implementation solution to realize MDD. If one would like to apply DSML-based techniques to airborne software development, the tools must support the traceability requirements posed by the development of these systems.

If the models express the low-level requirements, the traceability between a model and the code basis is required. Most often, however, models are modified, simplified, or processed in various ways before the code generation begins. These modifications are implemented with model transformations. This means that not only model-to-code traceability must be supported, but model-to-model traceability as well. These traceability links must be established between the input and the output model of a transformation.

This is not a simple task, because the traceability represents and requires domain knowledge along with an understanding of the transformation. In most cases, traceability links cannot be established without human effort.
2. BACKGROUND AND RELATED WORK

Traceability in software engineering often deals with requirements traceability [7][4], which is defined as the ability to describe and follow the life of a requirement in both a forwards and backwards direction. Requirements traceability is also aimed at keeping stakeholder requirements aligned with system evolution. [11] uses empirical studies to decide which kinds of traceability links are most important and then describes the design of reference models for traceability based on these links. These reference models capture the basic elements needed for traceability and can be specialized to provide particular traceability support to different applications.

[19] describes a tool called Unified Transformation Infrastructure (UniTI) that implements a semi-automatic notion of traceability. They focus on model-to-model transformations, which the authors claim can be used for text-to-model or model-to-text transformations if the text is also expressed as a model. The tool offers generic traceability, in which every model element can be traced through general links that are generated automatically, while links specific to a given transformation are also supported.

Another tool, [9], focuses primarily on tool integration and uses a metamodel based approach to traceability. The user defines metamodels for the different tools to be integrated along with a link metamodel that specifies the elements of the integrated tools that are related. Additionally, tool adapters must be manually defined for each tool so that they can interface with the generated traceability information. By using triple graph grammars, the traceability links are automatically included in the specification of the transformation rules. The correspondence part of the triple graph grammar rules is related to our traceability link specification in our rewriting rules.

Requirements traceability has also been used in MDD. [18] describes a framework for web applications in which a model transformation is used to obtain a navigational model automatically from a requirements model. A custom tool then uses the generated navigational model to generate a traceability report linking the navigational model with its requirements model.

The work described in [6] also uses traceability in model-driven development, but in a more general sense. They note that a variety of artifacts and models are used to describe a system throughout the MDD process and define traceability links to make the relationships between these different artifacts explicit in interconnection graphs called traceability models. The traceability models are then used in tools for interactive consistency management.

Traceability in triple graph grammars [14] is explored in [5]. There, triple graph grammars are used to perform incremental model transformations for synchronization purposes. A correspondence metamodel defines the mapping between a source and target metamodel, and an instance of this correspondence model is then defined and captures the links between source and target model elements. This correspondence model stores the traceability information that is later used to preserve the consistency between two models. Our approach to specifying the links between source and target model elements is similar to this, but whereas [5] defines a correspondence model that resides in between the source and target models in the transformation, we have added the traceability concepts directly to our transformation language.

The ATLAS Transformation language [3] is another model transformation tool in which traceability has been explored. [8] presents a simple traceability metamodel that allows traceability elements to be created in the same way that other target model elements are created. This approach is similar to our proposed solution.

A more general solution for traceability is presented in [20], including a relatively large traceability metamodel and a conceptual solution for tool integration. The metamodel and framework are tool and domain independent. Traceability specific to model to text transformations is described in [10]. The authors there use a metamodel to describe the links between source model elements and generated code and describe the implementation in a tool called MOFScript [15].

The Epsilon Generation Language [12] (EGL) is a template-based language that generates textual code from graphical models. The code to be generated is divided into sections. EGL provides a traceability API that makes the information of the actual template execution accessible.

A thorough survey of software traceability is given in [16], along with reviews of different frameworks, representations and approaches to traceability. [1] provides an in-depth description of a tool that provides an end-to-end framework for traceability.

The authors of [16] note that one of the main reasons
for the limited use of traceability is that tools and environments fail to provide support for all of the types of artifacts constructed during the software development life-cycle. Our tool tries to fill part of this gap in MDD by generating traceability information based on given traceability specifications during certain parts of the process. Specifically, we target the generation of traceability links between the input and output models of a transformation as well as traceability links between an input model and code generated by a code generator.

3. A TRACEABILITY SOLUTION

The overall architecture of our solution is depicted in Fig. 1. The diagram is organized around two main steps. (i) We have an input model, and an output model is generated from this input via model transformation. (ii) Furthermore, from the output model, we generate source code text. In the terminology of the recommendations for airborne systems, the input model corresponds to the low-level requirements, and the final output is the textual code.

It is clear from the concepts that we must establish traceability between the input model and the output model as well as between the output model and the code. This information will be stored in the Model-Model Traceability Data and in the Model-Code Traceability Data, respectively. The model-to-model traceability data is represented in graph-oriented format, whereas the model-to-code information is textual. In accordance with the recommendations, we need to produce traceability between the input model and the related sections of code. This is performed by a custom traceability tool.

We illustrate our approach on a simplified example. The input model of a transformation is a Signal Flow model. Since it is a DSML, it is defined by its metamodel (Fig. 2) specified in a syntax based on UML class diagrams.

A signal flow model may contain the following elements. An _InputSignal_ represents a signal that is processed by a signal processing unit. An _OutputSignal_ is the result of the processing operation. Signal processing components can be organized into hierarchies, which reduces the complexity of the model. A signal processing unit can either be _Primitive_ or _Compound_. A _Primitive_ can contain only atomic elements, while a _Compound_ can contain _Primitive_ or _Compound_ processing units. Signals are connected via ports: either the processing units have ports, or there may be local ports within compound components.

In our example model (Fig. 3), _Preprocessing_ and _Controller_ are compound processing units, whereas _Filter1_, _Filter2_, _ControlAlgorithm_, and _DAC_ elements are primitive signal processing components. The input signals and the output signals cannot be connected directly: they need an intermediate _LocalPort_.

Our example begins with a hierarchical signal flow modeling language and defines a transformation targeting a non-hierarchical signal flow language. This transformation may be useful for several reasons, but the main motivation is usually execution-related: if one wants to generate a low-level implementation for a signal flow, simulation engines often do not support the concept of hierarchies. The target metamodel of the transformation is a “flat” actor-based language without hierarchy, shown in Fig. 4.

![Figure 3: Signal Flow Example Model](image)

![Figure 4: Flat Signal Flow Metamodel](image)
create a RootContainer element and top level Queues for Ports. The block that is recursively called to flatten the hierarchy is expanded on the second line of rules in Fig. 5. The first rule on the second line creates top level Queues for each LocalPort in the input model. The third line of rules in Fig. 5 is responsible for creating temporary associations so that the hierarchy can be flattened. The transformation rule named FilterPrimitives is a conditional block that sends nested CompoundComponents back through the recursive rule and sends all of the regular Components to the final row of rules. This final row of rules is responsible for creating the Actors in the output model, along with their Receivers, Transmitters and the connections between them.

Note that because of the several types of connection classes in the original meta-model, four rules are needed to deal with translating these into the target model, which are the first four rules in the third row of Fig. 5. The transformation contains a total of twelve transformation rules, two test cases, and one recursive rule.

In order to address traceability, we have extended the transformation language with a construct called the TracesTo connection. It is quite intuitive in our example that, in the rule CreateActor, the current PrimitiveComponent must trace to the newly created Actor, as it is described by the TracesTo relationship. In our experience, this is a simple yet effective way to define traceability between the input model to the output model of a transformation. Note that this new relationship extends the semantics of the transformation language.

Conceptually, the structure of the transformation rule directly corresponds to the structure of the model-to-model traceability data. The basic entities of this correspondence are the rule realizations with the actual bindings to the source and target models. When the rule in Fig. 6 is executed multiple times, all the elements are bound to specific model elements that may (and some of the elements do) differ for each rule execution. A rule realization contains the information regarding which model element traces to which model element as defined by the bindings of the source and target of the TracesTo relationship. The rule realization entities of the traceability data are organized according to the control flow realization. Similarly to the rule realizations, a control flow realization means the actual order of the rule realizations. If there are no loops and recursion, the control flow realization is equivalent with the control flow itself. These concepts are directly reflected by the metamodel of the model-to-model traceability data (Fig. 7).

The ExecuteRule represents a rule realization. It is organized into sequences with the Sequence connection. UdmObject represents a model element (the name originates from the Universal Data Model 'UDM' tool that is the imple-
Figure 7: Metamodel of the model-to-model traceability data

In our implementation, we simplified the conceptual design: we do not store the relations between the model elements and the rule elements to which they are matched. The unique IDs refer to the actual model elements, and the $dataNetworkId$ corresponds to the model to which the given model element belongs to, since the transformation engine works on multiple models.

The code generator is usually implemented in an imperative programming language. Based on the model elements it is generated from, the code is divided into sections that are identified by an ID. The model-code traceability data relates the ID of a model element to the ID of the section generated from it. Conceptually, this relation can be handled as another $TracesTo$ relationship.

The traceability tool computes the transitive closure of the $TracesTo$ relationship, and provides a user interface facilitating the traversal of the transitive closure by highlighting the code or the model element. This is illustrated in Fig. 8.

Figure 8: Navigation between model and code

4. CONCLUSIONS

Software development for airborne systems has always been a mission critical activity. The most up-to-date recommendations suggest strong traceability requirements: high-level requirements must be traced to the low-level ones that must be traced to source code units. In a simple manner, this means that if one picks an arbitrary line in the source code, the corresponding requirement(s) implemented in that line must be shown.

We have implemented a tool, which shows the low-level requirements to which a certain source code block traces back to by selecting the given source code block. To our best knowledge, there is no tool that gives a support for such functionality.

In our approach, the tool requires two types of changes. (i) When specifying model-to-model transformations, traceability links must be specified by the developer of the transformation. (ii) The model-to-code transformation must create identifiers for the code blocks, specify them by their positions within the text, and finally provide the correspondence between the model element IDs and the block IDs. This solution is inherently cross-domain, and it connects modeling projects with textual projects.

Although we used a simple example in the paper, we and our supporters have built several models that have been used to advance and automate avionic software development. The implementation of our approach uses the GReAT toolset, but the concepts and ideas discussed in this paper can be transferred to any tool applying graph rewriting-based formalism for model transformation.

The most important direction of future work is to examine the possibility of generating the traceability links in the transformation rules automatically. If this issue is solved we expect the major model transformation tool to support traceability as well.

5. ACKNOWLEDGMENTS

This work was sponsored by DARPA, under its Software Productivity Program. The views and conclusions presented are those of the authors and should not be interpreted as representing official policies or endorsements of DARPA or the US government. Approved for Public Release, Distribution Unlimited.
6. REFERENCES


