Getting Back to Basics: Promoting the Use of a Traceability Information Model in Practice

Patrick Mäder[1], Orlena Gotel[2] and Ilka Philippow[1]

[1]Department of Software Systems
Ilmenau Technical University, Germany
patrick.maeder|ilka.philippow@tu-ilmenau.de

[2]Department of Computer Science
Pace University, New York, USA
ogotel@pace.edu

Abstract

It is widely assumed that following a process is a good thing if you want to achieve and exploit the benefits of traceability on a software development project. A core component of any such process is the definition and use of a traceability information model. Such models provide guidance as to those software development artifacts to collect and those relations to establish, and are designed to ultimately support required project analyses. However, traceability still tends to be undertaken in rather ad hoc ways in industry, with unpredictable results. We contend that one reason for this situation is that current software development tools provide little support to practitioners for building and using customized project-specific traceability information models, without which even the simplest of processes are problematic to implement and gain the anticipated benefits from. In this paper, we highlight the typical decisions involved in creating a basic traceability information model, suggest a simple UML-based representation for its definition, and illustrate its central role in the context of a modeling tool. The intent of this paper is to re-focus attention on very practical ways to apply traceability information models in practice so as to encourage wider adoption.

1. Introduction

Traceability is said to bring many benefits to a software development project, including the demonstration that requirements have been satisfied [1]. That is one reason for requiring it in the development of safety-critical systems or to reach a higher certification level according to a maturity model like CMMI. A major drawback of traceability is the high effort associated with creating and updating relations. To make traceability attractive for practitioners who are either ambivalent or required to do it, there are two possible tactics. First, provide an easier and less effort-intensive approach for creating and updating traceability relations; second, increase the benefits of traceability to compensate for its costs. Both approaches can be supported by implementing a traceability information model. Aizenbud-Reshef et al. [1] refer to the optimal traceability information model as one that is customizable and extensible by the user.

At its most basic, traceability is achieved by establishing a relation between two artifacts. With semantics given to such relations, the relations can be validated and various analyses undertaken. Several researchers have proposed different traceability meta-models or reference models, each defining artifact types and relation types, based upon studies with practitioners [9], [10]. Nevertheless, recent interviews conducted by the authors of this paper with practitioners working in different domains have indicated that traceability meta-models are still rarely defined and used [6]. One reason that was identified was the lack of knowledge about the benefits that use of such a model would bring. Another reason, cited by more experienced practitioners, was the limited support for creating and customizing such models within current development tools.

This paper is targeted at this addressing these issues. Traceability information models are not a new concept, as shown in Section 2, so we seek to highlight some of the issues that potentially limit widespread adoption in this section. In Section 3, we propose a simple way to define a traceability information model, and in Section 4 we discuss how to define constraints for traceability relations so as to support traceability handling and analyses. In Section 5, we provide a short guide of heuristics to consider when creating an initial and viable traceability information model. The approach, as proposed in this paper, has been implemented and successfully used in our traceMaintainer prototype [5] as the basis for the (semi-)automated traceability updates it performs, and this is summarized in Section 6. Adoption and integration with other development tools is also discussed.

2. Related Work

Ramesh and Jarke [9] made extensive observations of traceability practice in several organizations. Based on

these observations, the authors distinguished high-end and low-end traceability users, each with different needs. The authors therefore proposed two levels of reference model, each for traceability with differing numbers of predefined relations and entities. The diversity of relation types, coupled with a lack of precise definition and achievable tracing goals, makes the application of the reference models a complex task. The work does not present how to implement a reference model in tools nor how to customize it in relation to project-specific needs. The only suggested mechanism to configure the model is to cut or add parts.

Spence and Probasco [10] present several traceability strategies as meta-models for the Unified Process. The authors use only one type of trace and do not discuss the implementation of their traceability meta-models.

Within tools, two different ways to implement traceability information models are commonly adopted. The first approach is trace tagging. This allows the user to define custom types for relations that are applied to a generic trace type available in the tool as attributes, stereotypes or tags. This is the approach that is supported by most commercial development tools, like IBM DOORS, Sparx Enterprise Architect and IBM Rhapsody. The second approach is to define specific relation types within the meta-model of the tool. Several research prototypes and commercial tools implement this approach and so provide for a fixed set of specialized traceability relations [8]. Both approaches have benefits and drawbacks. The trace tagging is easily customized by the user within the tool, but the tool generally treats all traces equally and is not able to validate relations based on the type or to provide specialized behavior and analyses based on the type without additional work. A traceability meta-model with specialized types of relations can provide for special treatment of those relations, but these are fixed in their application, while specific project needs require customized types of relations.

Based on the drawbacks of the two predominant approaches underlying tools, it is clear that neither provides particularly effective traceability support in all cases. Projects may have very specific constellations of artifact types and, at the same time, demand user-friendly, sophisticated solutions to mitigate the labor-intensive activity of establishing and maintaining traceability. The identification of traceability information (artifacts) that the stakeholders intend to capture and to use should inform the traceability process for a new project. The reduction of artifact and relation types to the minimal necessary in order to achieve certain traceability analyses should be one of the main goals of this definition activity.

Letelier [4] proposes an interesting way to implement a traceability meta-model by extending the UML meta-model. The definition of traceable entities and traceability relation types between them is based on a UML profile. The approach has several benefits: using the available extension mechanisms of the UML to define the meta-model; storing traceability relations as part of the development model itself; and allowing the definition of project-specific types of traces between certain artifacts. There are also several shortcomings. First, the existing trace relation within the UML meta-model does not allow to trace all types of UML entities (e.g., attributes, methods and relations). Taking the UML trace relation as the base type for all defined relations means that these relations inherit the same shortcoming. On the other hand, the original trace relation still exists after specialization and can be created between all supported UML entity types, requiring the user to decide for the appropriate relation type in the relevant context. Also, constraints as to the minimum number of trace relations between certain entity types are not supported, and constraints are needed in order to validate the existence of relations.

What is lacking is an approach that effectively helps practitioners to create and then implement a project-specific traceability meta-model (what we refer to as a traceability information model) within development tools. Practitioners must be able to define such models according to their traceability needs and budgets, with ease and using a language with which they are familiar. Starting from very basic traceability information models, there needs to be the potential to elaborate and customize these models as necessary. The traceability information model also needs to be described in such a way as to guide both the set up and traceability analyses made possible in the development tools that are used.

3. A Traceability Information Model

A basic traceability information model consists of two types of entity, traceable artifacts and traceability relations between these artifacts. It also defines which types of artifacts are intended to be traced to which related artifact types and by what type of traceability relations. There are several reasons for implementing traceability in an agreed way like this:

- it ensures consistent results in projects with multiple stakeholders;
- as traceability is also used by people who did not create it, these people need to know how it has been defined and what to expect from it;
- as tracing is a complex task, a traceability information model provides a guideline to ease its set up and allows for the validation of changes;
- coverage analysis only becomes possible after having defined what the expected coverage is;
- a traceability information model is a necessary precondition for automated traceability handling, validation and analyses [7].

A traceability information model can be represented in many ways. Representing it as a graphical model in a description language that is familiar to most people in the area of software development makes it easier to understand and customize. We propose the use of UML as it is the standard for the modeling of object oriented systems. Most practitioners in the software development field are at least able to read and customize UML diagrams. UML is often used to support the development activities within a project itself, so use for defining a traceability information model requires no additional knowledge and tools.

In this sub-section, we step through how traceability can be precisely defined by creating a traceability information model. In Section 4, we refer to the definitions and show how each facilitates certain analyses and validation.

Traceable artifact types and permitted traces The traceability information model consists of entities representing types of traceable artifacts. A precondition for a traceable artifact is a unique identifier. Each traceable artifact is represented as a class. Examples for traceable artifact types are requirement, test case and design component. Valid traces between two traceable artifact types are defined as a relation between them. Only defined traces are allowed to be created within the project. Figure 1 shows that use cases are allowed to be traced to test cases.

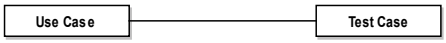

Figure 1. Permitted trace

Count of required traces Often, a minimum number of traces between two traceable artifact types is required. By using cardinalities for the relation between two traceable artifact types it is possible to require a number of relations between instances of both types within the project. Figure2 shows a scenario in which each use case has to be traced to at least one test case and vice versa.

Figure 2. Minimum count of required traces

By defining concrete cardinalities for a relation, instead of an interval, it is possible to require an exact number of relations between two traceable artifacts. Practitioners developing safety-critical systems and aiming for certification in our interviews reported this need. They have to prove that every requirement has been implemented by an exact number of implementation artifacts.

Trace types and related artifact roles By defining a name for a relation between two traceable artifact types it is possible to define types of traces. This additional information allows for a better understanding of a trace's purpose and for more concrete descriptions in reports and tools. Furthermore, the type can be used to group similar relations between different traceable artifacts for analysis. In addition to providing trace types, role names for each related traceable artifact type can be attached in order to allow for clearer documentation and description of traces. Figure 3 shows that requirements require test cases, while test cases validate requirements. Between both artifacts there exists an validation relation.

Dependency between related artifacts By creating dependency relations between two traceable artifact types, a dependent and an independent artifact are defined. This means that the independent artifact has influenced the creation of the dependent one, but not vice versa. This information can facilitate a more efficient determination of the impact of a change to an artifact and the propagation of this change to related artifacts. Figure 3 shows a test case that is dependent upon the requirement it is validating.

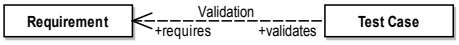

Figure 3. Defined dependency, type and artifact roles for a trace

Mapping between tool and project artifact types Tools provide generic artifact types from the modeling domain they support. Within a project, the tool artifact types are often used to represent more specific project artifact types. It is also possible that one tool artifact type represents multiple project artifact types or multiple tool artifact types are used to represent the same project artifact type. For example, a requirement artifact of the tool can be used to represent user and system requirements in a project (as per the Figure). In a more generally defined information model, a requirement artifact within the project could be permitted to be created as either a use case or a requirement artifact within the tool.

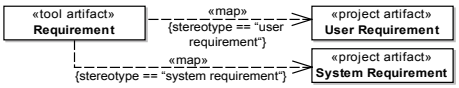

Figure 4. Mapping between artifact types

The definition of a mapping shows which tool artifact types are intended to be used to represent project artifact types. If the traceability information model is used by a tool it is absolutely necessary to define such mappings in order to allow for automated use of the model. The defined tool artifact types do not necessarily have to belong to only

one development tool. A stereotype or attribute can define the tool an artifact type belongs to.

For a mapping of one tool artifact type to multiple project artifact types (as per the Figure), it is necessary to define additional constraints that allow the corresponding project artifact type for a given concrete artifact to be determined. These constraints can be expressed in the Object Constraint Language (OCL). The constraint could, for example, evaluate the stereotype of an artifact, the name of the package it is contained in or the tagged values attached to it.

4. Analyses Based on a Traceability Information Model

A defined traceability information model enables differing analyses, a selection of which are illustrated in this section. Guidance to practitioners is to first understand the analyses that are likely to be the most useful to the project at hand and to create the simplest possible traceability information model to enable these analyses. More sophisticated types of analyses may require subsequent elaboration of the traceability information model, so such customization needs to be possible.

Validating traces Validation of traceability is important in order to ensure the correctness of any analyses based on it. There are two ways to ensure that a project's traceability relations conform with the defined traceability information model. First, by triggering an analysis to check at a certain point in time. It is important to note that only the conformity to the model can be validated by this analysis. It is not yet possible to find relations that are semantically incorrect. Second, allowing a trace only to be created (either manually or automated) if it is permitted by the traceability information model. The second way ensures that no false relations exist at any time and should be supported by tools where this is a critical constraint for analyses. Nevertheless, a validation on demand and triggered by the user is also important to support. For example, if the traceability information model is changed after a set of traces has already been created, a number of traces may become invalid.

Impact analysis and change propagation Facilitated by dependency relations between artifact types defined in the traceability information model, impact analysis and change propagation provide more specific results as they can be constrained to only refer to dependent artifacts. For example, so that changes to design elements are only propagated to dependent code elements, instead of also to independent requirements.

Coverage analysis Within the traceability information model, permitted traceability relations between artifacts are defined. By defining cardinalities, a certain number of relations can be required and this can be validated by a coverage analysis. This analysis is not only thought to find missing traces, but even more to find missing artifacts. For example, requirements that have not yet been implemented. By also searching for those elements within the model that are not yet traced but have the same types as those of the coverage analysis, possible missing relations can be suggested. The results of the coverage analysis can also provide information about the status of a project's traceability set and so about the comprehensiveness of other analyses.

Relation count analysis Another possible analysis is to check the number of existing relations between two types of artifacts. A large number of relations between two types of artifacts can suggest either that the trace granularity is too coarse or that functionality or responsibility is concentrated in one artifact. Analysing the statistical distribution of trace counts may help to decide between both cases. An example of an analysis result suggesting a design problem could be a class with five traces to different use cases. This "super class" may have too much functionality concentrated in itself. If all classes are related by many traces this may indicate that the tracing to classes may be too coarse and that an adjustment of the traceability information model would be useful.

5. Creating a Traceability Information Model

Traceability information models are not a new idea and the analyses they make possible would seem to offer value. However, it is evident that even where the benefits are recognized, the matter of providing simple practitioner guidance and support still needs addressing.

It is clearly necessary to start by analyzing the needs to be satisfied by traceability in a given project. This requirements analysis helps to identify relevant artifacts to be retrieved in a trace inquiry. In addition, the development methodology and functionality of the development tools used define available and accessible artifacts, so also influence the creation of the traceability information model.

Some prior consideration about the granularity and types of related artifacts can avoid inconsistencies later and reduce the number of relations established, resulting in less effort for traceability handling. Usually, a high-level requirement is taken as the starting point for traceability in a traditional development process (post-requirements traceability [3]). From there, artifacts are traced to one or more end points in a forward and backward direction. Intermediate artifacts (e.g., analysis models), should also be related if the intention is to keep them consistent while other artifacts evolve. If intermediate artifacts are traced, then usually no other trace should bypass these intermediate artifacts in order to avoid inconsistencies.

Establishing the 'right' granularity of traced artifacts depends on the intended use and on the available resources for handling traceability. For example, it might be sufficient to trace a low number of features to their corresponding imple-

mented components in order to guide new team members to the relevant part of the code, but it could also be required to document that each system requirement has been implemented, demanding finer-grained traceability. While there is no standard answer to what is the 'right' granularity, some points may be worth considering: (1) The granularity of related artifacts should correspond to one another. Tracing coarse artifacts on one side to finer-grained artifacts on the other side requires a large number of relations while not providing more information than tracing coarse artifacts on both sides (e.g., tracing features to methods requires a larger number of traces while likely not providing more information than tracing features to classes or components). (2) It usually makes little sense to trace several artifacts that are in a hierarchical relation to each other as the relation to the coarser artifact can be derived from the finer-grained one (e.g., tracing features to classes and use cases to methods, while features are already related to use cases).

6. *traceMaintainer* Prototype

Our traceMaintainer prototype reads a traceability information model defined according to the description in Section 3 and stored in XMI format. Advantages are that most UML tools are able to read and write models in the XMI format. Furthermore, the format allows users to customize the traceability information model within the tool that she or he is used to and without the need for the tool vendor to provide a special editor.

The traceability information model is used by traceMaintainer [5] to validate any intended new traceability relation regardless of whether created manually by the developer or as part of a (semi-)automatic traceability update. Furthermore, traceMaintainer provides change propagation after changes to related artifacts by setting the status of related relations to suspect. Changes are propagated to traced artifacts and only to those defined as dependent on the changed one within the information model (see Section 3).

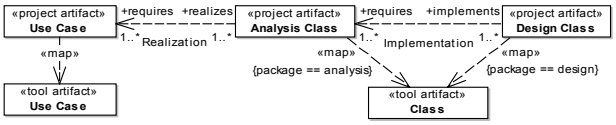

Figure 5. A simple information model

The figure shows a simplified traceability information model that we used during an experiment performed to explore traceMaintainer's capabilities. The model was created with Sparx Enterprise Architect within a few minutes and saved in XMI format. The subjects were provided with a printed version of the traceability information model to inform them about the intended traceability for the project. The experiment required the subjects to create traces between use cases and analysis classes and traces between analysis classes and design classes. Most subjects reported that traceMaintainer prevented them from creating traces inappropriately and that they liked the guidance provided when working on establishing traceability in the experiment. Furthermore, the analysis of the experiment became easier as we could be sure that only traces according to the traceability information model had been created, and we could also have more confidence in the analyses provided.

7. Conclusions and Future Work

A traceability information model is the baseline for any analyses based on traceability. This paper has illustrated how a basic traceability information model can be presented and used in practice, and re-iterated the benefits its use can bring. Implemented within development tools, it has the potential to ease traceability creation and maintenance, and to bring more confidence to use. The practitioner simply has to create the traceability information model via simple description and, after referencing it within the used tool, instant validation of trace creation and changes can be made, and desired analyses become supported.

Traceability information models are an essential component of any traceability process and this paper is an attempt to get back to some basics to both encourage and facilitate wider adoption of these models in practice. In cooperation with EXTESSY AG [2], we are currently discussing the implementation of traceability information models according to this approach in the context of tool integration, so as to support traceability amongst different tools and broaden the access to practitioners.

References

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

[2] Extessy AG, Wolfsburg, Germany. `www.extessy.com`.

[3] O. C. Z. Gotel and A. C. W. Finkelstein. An analysis of the requirements traceability problem. In *First Int'l Conf. on Req. Eng. ICRE*, pages 94–101. IEEE CS Press, 1994.

[4] P. Letelier. A framework for requirements traceability in UML-based projects. In *1st TEFSE*, UK, Sept. 2002.

[5] P. Mäder, O. Gotel, and I. Philippow. Rule-based maintenance of post-requirements traceability relations. In *Proc. 16th Int'l Req. Eng. Conf.*, Barcelona, Spain, Sept. 2008.

[6] P. Mäder, O. Gotel, and I. Philippow. Motivation matters in the traceability trenches. submitted, 2009.

[7] F. A. C. Pinheiro. Requirements traceability. In *Perspectives on Software Requirements*, pages 91–113. Kluwer Academic Publishers, The Netherlands, 2004.

[8] F. A. C. Pinheiro and J. A. Goguen. An object-oriented tool for tracing requirements. *Software*, 13(2):52–64, Mar. 1996.

[9] B. Ramesh and M. Jarke. Toward reference models of requirements traceability. *IEEE TSE*, 27(1):58–93, 2001.

[10] I. Spence and L. Probasco. Traceability strategies for managing requirements with use cases. WP TP166, IBM, 2000.