The Journal of Systems and Software 85 (2012) 2205–2227

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Controversy Corner

Towards automated traceability maintenance

Patrick Mäder[a,*], Orlena Gotel[b]

[a] Institute for Systems Engineering and Automation (SEA), Johannes Kepler University, Linz, Austria
[b] Independent Researcher, New York, USA

ARTICLE INFO

Article history:
Received 18 November 2010
Received in revised form 27 July 2011
Accepted 19 October 2011
Available online 25 October 2011

Keywords:
Event-based development activity recognition
Model changes
Requirements traceability
Rule-based traceability maintenance
Software system evolution
Traceability decay
Traceability maintenance

ABSTRACT

Traceability relations support stakeholders in understanding the dependencies between artifacts created during the development of a software system and thus enable many development-related tasks. To ensure that the anticipated benefits of these tasks can be realized, it is necessary to have an up-to-date set of traceability relations between the established artifacts. This goal requires the creation of traceability relations during the initial development process. Furthermore, the goal also requires the maintenance of traceability relations over time as the software system evolves in order to prevent their decay. In this paper, an approach is discussed that supports the (semi-) automated update of traceability relations between requirements, analysis and design models of software systems expressed in the UML. This is made possible by analyzing change events that have been captured while working within a third-party UML modeling tool. Within the captured flow of events, development activities comprised of several events are recognized. These are matched with predefined rules that direct the update of impacted traceability relations. The overall approach is supported by a prototype tool and empirical results on the effectiveness of tool-supported traceability maintenance are provided.

1. Introduction

Traceability provides for a logical connection between artifacts of the software development process (Gotel and Finkelstein, 1994). In support of change management tasks, traceability delivers important information about the possible consequences of a changing requirement. For project management tasks, traceability supports the control of a project's progress and provides a way to demonstrate the realization of user requirements. Traceability is essential for numerous quality-oriented software development practices such as these.

Though widely accepted as beneficial, the costs associated with traceability can be high, so the return on investment remains debatable (Arkley and Riddle, 2005; Egyed et al., 2007). Unless mandated, traceability is rarely used throughout all development stages, due firstly to the number of artifacts or elements therein that often need to be related to yield value, and due secondly to the need to maintain these relations each time a change occurs. Even where the set of relations is minimal, the maintenance of traceability demands effort. While attention has been directed toward approaches for establishing traceability initially among artifacts, less attention has been paid to ensuring this traceability remains correct over time. This is the problem of traceability decay and is the focus of this paper.

The maintenance of traceability relations is a multi-step activity. As changes occur to the artifacts of software development, it is essential to appreciate both where and how these artifacts play a role with respect to the current traceability, along with an understanding of the encompassing development activity that can characterize the nature of the change. It is then necessary to understand the impact of the development activity on the traceability and to carry out those activities that can re-establish the traceability, at least to the prior levels. These core tasks demand effective method and tool support. This paper describes a novel approach for the maintenance of requirements traceability relations. The approach currently supports development models expressed in structural United Modeling Language (UML) diagrams and converts part of the manual effort necessary for traceability maintenance into computational effort. There are two important innovations with the approach: first is the automatic identification of development activities with impact on existing traceability relations (event-based development activity recognition); and second is the use of rules to describe development activities and the necessary updates in an abstract way (rule-based traceability maintenance). The approach is (semi-) automated as, depending on the nature of the change and the status of the existing traceability, the user may have to provide input to the process.

In this paper, we provide an exhaustive and mature description of an approach that we have developed over the past several years.

* Corresponding author.
E-mail addresses: patrick.maeder@jku.at (P. Mäder), olly@gotel.net (O. Gotel).

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.10.023

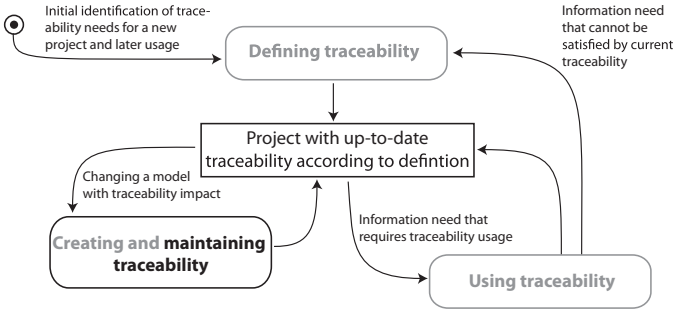


Fig. 1. Traceability life cycle for a project

Certain parts of the approach have been published previously. In three prior publications we discussed aspects of the approach, Mäder et al. (2008a) gave an initial overview of the approach, Mäder et al. (2008b) discussed technical details of one particular component of the approach, namely the development activity recognition, and Mäder et al. (2009a) referred to link update concepts and introduced different types of development activities according to the required update. In addition, tool demonstration papers provide for an overview of the development prototype. The most complete and up-to-date tool information is provided in Mäder et al. (2009b). The objective of this current paper is to consolidate the work into one primary publication at the requisite level of detail. Building upon a thorough analysis of the state of the art in the field of traceability maintenance, the current paper demonstrates and discusses how a (semi-) automated approach in this topic can convert large parts of tedious and error-prone manual effort into computational effort.

The paper is organized as follows. The topic of traceability maintenance and related research is discussed in Section 2. Section 3 provides a conceptual overview of the entire approach, outlining its scope, assumptions and phases. Sections 4 and 5 provide depth on the two main phases of the approach, and an evaluation of the approach is described in Section 6. The paper concludes with a critical review and suggestions for future research in the area.

2. Traceability maintenance

Providing traceability for a project is not a trivial matter; different activities are necessary to both create and then maintain traceability relations, as suggested in Fig. 1. An agreed methodology for traceability, specifying how to create, maintain and use traceability, is not generally available (Aizenbud-Reshef et al., 2006). An important reason for this absence is the high variability in development processes used in practice. Nevertheless, common to all processes is the necessity to specify which artifacts should be related and how this information should be used to obtain a consistent set of traceability relations across developers (Dömges and Pohl, 1998).

Pinheiro (2004) divides the 'production' of traceability relations into perception, registration and maintenance. Other authors refer to the registration as creating, establishing or installing traceability relations. These terms will be used interchangeably in this paper. Likewise, traceability maintenance and update, and traceability relation and link, will be used synonymously.

In recent years, much research has been dedicated to techniques for the automated identification and creation of traceability relations. The majority of these approaches apply text mining and information retrieval techniques to identify candidate relations (Alexander, 2002; Antoniol et al., 2002; Marcus and Maletic, 2003; Hayes et al., 2003; Lucia et al., 2008). Even with these emerging techniques, manual intervention to prune candidate relations cannot be completely avoided. One day it may be viable to simply rely on automated trace generation on demand and as needed, but that requires substantive advances in the precision of these techniques to remove the need for continual re-confirmation of the candidate relations. An alternative is to generate a quality set, through these techniques and manual pruning, and then to focus on maintaining them. This paper focuses on the latter strategy. There has been less research work on the automated maintenance of traceability relations. Maintaining traceability means to prevent its decay while related artifacts evolve. Aizenbud-Reshef et al. (2006) refer to maintenance as the most challenging aspect of traceability.

Murta et al. (2006) characterize the problem of traceability maintenance between architectural elements and source code as follows: "...given an initial set of established traceability links, and given that both an architecture and its implementation can evolve independently, how can traceability links be updated with the addition of new links, removal of existing links, and changes in existing links to ensure that each architectural element is at all times accurately linked to its corresponding source code configuration items, and vice versa?" Without maintenance, traceability relations between elements get lost or represent false dependencies. Such a step by step degradation of traceability relations leads to traceability decay. This can be prevented by continuous or on-demand traceability maintenance. On-demand maintenance offers the theoretical benefit that relations are only updated according to the current state of the model, with potentially fewer incremental update steps as compared to continuous maintenance. On the other hand, the demand for updated traceability might arise a long time after the change to the model that has caused the need for maintenance and it might be harder to perform than instantaneous continuous maintenance. From a theoretical point of view, both options have advantages and disadvantages, highlighting the need for further empirical studies in this area. It would be important to independently assess the quality of the traceability relations established, as a result of following the two strategies, with respect to a shared set of traceability-enabled tasks demanding impact analysis and change management. Moreover, the value of blending continuous and on-demand approaches to maintenance suggests an area of open research.

This section explains why traceability maintenance becomes necessary during the development and evolution of a software system. It also discusses the strengths and weaknesses of existing approaches to the problem in order to put the approach proposed in this paper into context.

2.1. Why traceability maintenance is necessary

A common way to cope with the complexity of software systems engineering is modeling the product to be developed at different levels of abstraction and from different perspectives. This process is called model-based development. A model can be defined as an abstraction of some real world object and, in the context of a development process, it refers to the product which is the subject of engineering (OMG, 2003, 2010). In software engineering activities, models can be used to represent the requirements, the design and the implementation of a software system. As all the models of one development project describe different aspects of the same product, they are interrelated. For example, the design of a software system depends on its requirements, while the implementation depends on its design. The (OMG, 2010) provides a set of structural and behavioral diagrams that allow many facets of a development process to be modeled. The UML is further supported by many modeling tools and is the quasi-standard in object-oriented development. An extension, the Systems Modeling Language (SysML) (OMG, 2008), provides additional diagrams and options for systems development.

Model-based development processes may be viewed as multi-phase transformation processes from the initial problem statement

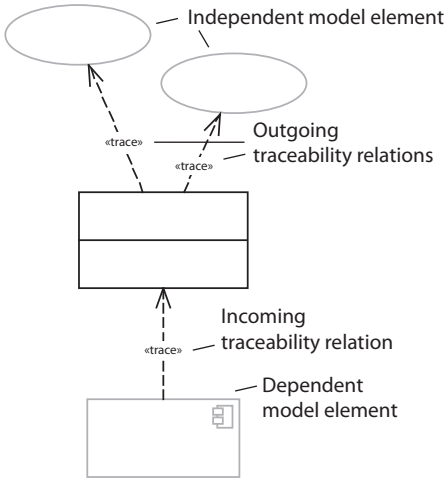


Fig. 2. Distinction of traceability relations into incoming and outgoing relative to the selected element within a model

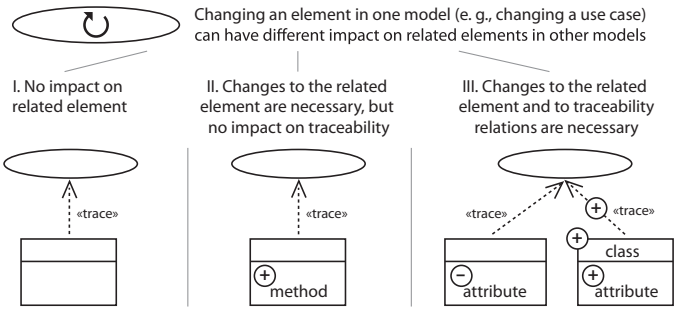


Fig. 3. Changing an element in one model can have different impact on related elements and on the existing traceability relations

to the final solution (Jacobson et al., 1999). These transformations are carried out as development activities. Each of these activities is applied to or influenced by various input artifacts and creates new or improved output artifacts. If this multiphase transformation process is carried out only once, entering each phase only when the preceding phase has been completed, it is a waterfall-like process (Royce, 1987). Most problems nowadays are too complex to be solved in this manner and state of the art development processes are iterative and incremental (e.g., the Unified Process; Jacobson et al., 1999). Moreover, development does not always proceed in a forward direction (forward engineering). Rather, it may also involve working in a backward direction (reverse engineering). Combining forward and reverse engineering results in round-trip engineering, where developers opportunistically mix both modes of development. This mode of development necessitates interrelated model changes.

Creating an explicit traceability relation between two artifacts can capture their dependency. Within the UML meta-model (OMG, 2010), the representation of traceability relations is considered as a type of dependency with a given direction. The direction of such a traceability relation points from the dependent model element towards the independent model element, as shown in Fig. 2. This directionality is intended to convey semantics, but does not prevent bi-directional use or navigation of the traceability relation. Arlow and Neustadt (2005) state that a change to the independent element (supplier) may effect or supply information needed by the dependent element (client) and that the client in some way depends on the supplier. A stereotype "*trace*" is applied to distinguish traceability relations from other dependencies that are part of the models (Arlow and Neustadt, 2005; Weilkiens, 2006).

A major problem that arises in model-based software development is ensuring that related models evolve consistently while the development proceeds (Huzar et al., 2004). Finkelstein et al. (1994) state that checking consistency between perspectives and the handling of inconsistency creates many interesting and difficult research problems. Traceability can support this issue by propagating changes that happen to an element in one model to all its related elements in other models (Aizenbud-Reshef et al., 2006).

Such changes to related model elements can also necessitate maintaining the relations to reflect all the initial dependencies between the evolved model elements after the change. Three types of impact can be distinguished (see Fig. 3):

(I) The change can be purely corrective with no impact on the related element. For example, correcting typos within the description of a use case.
(II) The change can have impact on the related element, but not require changes to its structure. For example, a new method within a class is required due to an enhanced use case. The change to the original element also requires evolving the related element.
(III) The change can have impact on the related element and, due to changes in the structure, also on traceability. For example, an attribute has to be extracted into a new class due to an enhanced use case. The change to the original element not only requires evolving the related element, but also retaining the traceability between the model elements.

2.2. Related work

This section describes the related work in the area, differentiated according to the predominant mechanism used to achieve the traceability maintenance. The approach described in this paper builds upon each of these mechanisms, so points of difference are highlighted in the descriptions.

2.2.1. Subscription-based approaches

Cleland-Huang et al. (2003) present an approach that can help maintain traceability called event-based traceability (EBT). The authors link requirements and other artifacts of the development process through publish-subscribe relationships stored in a central database. Changes to requirements are categorized by seven kinds (create, inactivate, modify, merge, refine, decompose and replace) and events are raised according to kind. The identification of these changes will be discussed separately in Section 2.2.3. Created events are published to an event server that sends notifications to subscribers of the changed requirement. These notifications contain detailed information about a change to facilitate the manual update process of the subscribing artifacts. This work discusses a sophisticated change propagation mechanism, enabled by traceability and change recognition (i.e., informing the owner of a related artifact with a detailed message about the identified change to a requirement and its type). The approach does not discuss the actual maintenance of impacted traceability relations, but the event generation aspect of the work has inspired the approach discussed in this paper and will be discussed in more depth in Section 2.2.3.

2.2.2. Rule-based approaches

Spanoudakis et al. (2004) present a rule-based approach for the automatic generation of traceability relations between documents, which specify either requirement statements or use cases

(in structured natural language) and analysis object models. A first kind of rule, Requirement-to-object-model rules, and a technique based on information retrieval are used to automatically establish traceability relations between requirements and analysis models. A second kind of rule analyzes the relations between requirements and object models to recognize intra-requirements dependencies and establishes these relations automatically. The approach requires the export of all supported artifacts into the eXtensible Markup Language (XML) format and the rules generate traceability relations for the exported state of the models. Due to the use of information retrieval, there is uncertainty within the recognized relations and limited support for developers with false recognition. The approach, in its current form, does not appear to support the maintenance of traceability relations following artifact evolution explicitly, but the approach proposes interesting ideas that could feasibly do so. Two ideas influenced the approach discussed in this paper. First, the use of extensible and customizable rules that describe properties of expected artifacts in an abstract way. Second, the idea of organizing rules in the style of event, condition, action and to store these rules in the open XML format to facilitate their customization by the user.

Murta et al. (2006, 2008) describe an approach called ArchTrace that supports the evolution of traceability relations between architecture and implementation. The use of the extensible Architecture Description Language (xADL) for the description of software architectures and the use of Subversion for the versioning of source code is required in the current form of the approach. The authors trigger a set of eight policies on committing a new version of an artifact (e.g., suggest traceability link to a more recent configuration item version if the user creates a traceability link to an older version). These policies mostly ensure the update of existing traceability relations on artifacts to new versions within the version control system. The concept of having a customizable set of policies (or rules) whose evaluation is dynamically triggered by change events (i.e., committing a new configuration to the configuration management system) is similar to the approach discussed in this paper. The fact that there are no policies that would allow for the recognition of structural changes to models (e.g., the replacing, splitting or merging of related elements) has been recognized as a shortcoming of the approach. These can be primary triggers for traceability maintenance, however, and so are addressed by the approach described in this paper.

Mens et al. (2005) describe an extension to the UML meta-model to support the versioning and evolution of UML models. The authors classify possible inconsistencies of UML design models and provide rules, expressed in the Object Constraint Language (OCL), to detect and resolve these. The approach transforms the models into a supported format, applies their rules and suggests model refactorings based on the results. While the authors discuss the necessity for traceability management and change propagation during the evolution of UML models, they provide no support for this scenario. In contrast to other rule-based approaches, the approach discussed in this paper uses rules to specify change patterns that occur during the evolution of related artifacts and require the maintenance of traceability relations.

2.2.3. Approaches based on recognizing evolution

Cleland-Huang et al. (2002) describe a concept for the recognition of change types applied to requirements as part of their EBT approach (see Section 2.2.1). These change types are used for the description of a recognized change during change propagation (called change events). The authors distinguish and capture seven types of changes to a requirements model as events, as listed earlier. All seven change types are composed of a sequence of four different change actions (i.e., create requirement, set requirement attribute, create link and set link attribute). The recognition of complex change types (e.g., merge, refine, decompose and replace) depends on the manual creation of traceability relations with a certain type between the original requirement and the newly created requirement(s) and, in certain cases, on setting an attribute of the initial requirement to the state *inactive*. The authors provide an algorithm that identifies the seven change types within a sequence of captured change actions. Furthermore, the authors suggest triggering the actual recognition process only for a completed user-defined session in order to minimize the risk of false recognition. The concept of observing incremental and elementary changes to a model, and the recognition of compound change activities, is similar to the approach discussed in this paper. Due to the scope of the EBT approach, it does not deal with the more complex task of recognizing compound changes to models, and focuses more on the manual creation of traceability relations instead of maintaining them.

Engels et al. (2002) present a classification of UML model refinements to preserve consistency during the evolution of UML-RT models (a UML enhancement for real-time systems). The authors identify three kinds of atomic modification: creation, deletion and update, and the focus is limited to four model elements: capsules, ports, connectors and protocols. The focus of this work lays on preserving and maintaining consistency between two models after incremental evolution steps. The approach allows, for example, to demonstrate under which conditions a modified deadlock-free model remains deadlock-free. The work does not show how atomic changes can be combined into the recognition of composite change activities with development intent, nor how to maintain consistency in these cases. The identified atomic modifications are similar to those identified during the development of the approach discussed in this paper and helped to substantiate their correctness and completeness.

Hnatkowska et al. (2003) specify behavioral refinements in UML collaboration diagrams and describe how these relate to structural refinements. The purpose is to establish refinement relationships between different abstraction layers. The authors provide a classification of nine simple class diagram refinements (e.g., adding a class, modifying an attribute, modifying a method, adding an attribute to a class, splitting a class into two classes with an association, introducing a successor of a class, adding an association, modifying an association and introducing an intermediate class). The authors mention possible tool support, but do not discuss how these refinements could be detected and require the developer to establish the relationships manually at present. The work provided input for the identification of model changes that require traceability maintenance in the approach described in this paper.

Maletic et al. (2005) describe an XML-based approach to support the evolution of traceability relations between models expressed in the XML (with no restriction to the content of the model). The authors also describe a traceability graph and its representation in the XML, independent of specific models or tools. They discuss the issue of evolution and propose to evolve traceability along with the models by detecting syntactic changes at the same level and type as the relations (e.g., textual links require textual change detection). The authors do not discuss how to detect these changes nor how to update the impacted traceability relations, but refer to their own work on the analysis of fine-grained source code differences and mention that this work could be applied to artifacts in the XML format as well. The observation that traceability should be maintained along with incremental changes to related models has inspired the approach discussed in this paper. There are techniques available, like the Eclipse EMF Compare Framework that can efficiently compute differences between models in the XML representation, making the proposed concept a viable solution for tools storing

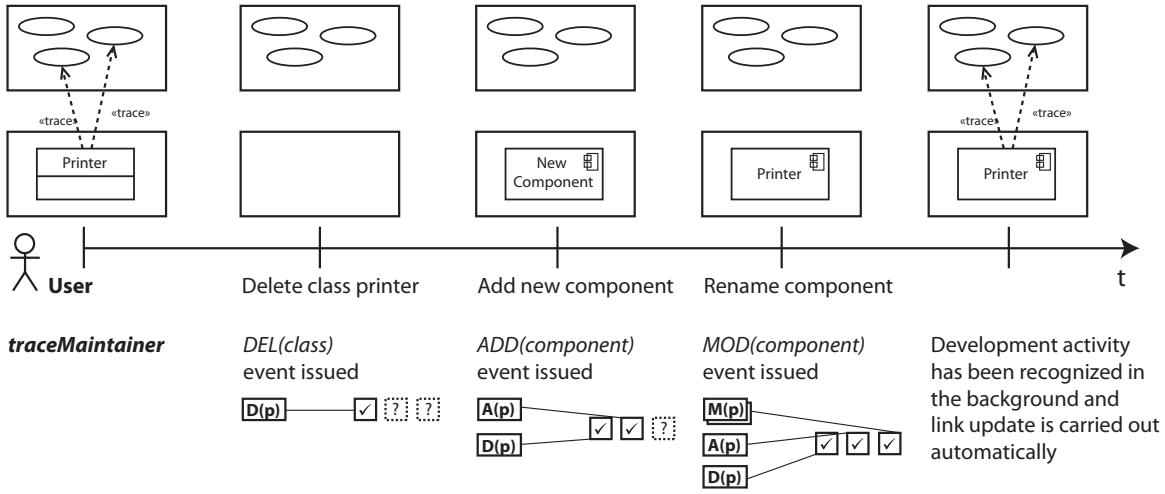


Fig. 4. Phases of the approach as visualized by the simple example of replacing a traced class by a component

their artifacts in the XML format. Nonetheless, artifacts are stored in many different formats and tools, causing the need for a conversion of changed models into the XML format anew after each change to the model in order to follow the proposed concept for those tools. The export of the complete model consumes time and would hinder the progress of the developer. For that reason we propose the computation of differences based on the original format of the artifacts.

Shen et al. (2003) suggest an extension to the UML meta-model via specified stereotypes according to four types of refinement (addition, deletion, connection and disconnection). The aim of their work is to support model modifications and, more specifically, model merging when different designers are concurrently working. Using these stereotypes on different abstraction levels of a project, the authors are able to check consistency between levels. Despite being able to maintain consistency of a model that is being evolved separately by multiple developers, using stereotypes may become a burden to designers who are changing the model and have to document these changes with the proposed stereotypes.

3. Overview of the approach

This section describes the development of a rule-based approach for the (semi-) automated maintenance of traceability relations based on the recognition of development activities. It describes the scope and two main phases of the approach. It also discusses the assumptions underlying the approach and the concepts developed to address a number of challenges. It ends with a technical overview.

3.1. Scope and phases

The approach is concerned with incremental changes to an evolving set of traceability relations, so the maintenance of already established traceability relations. The approach is not concerned with creating an initial set of relations, which is mostly the domain of techniques based on information retrieval and data mining. Section 2.1 highlighted contemporary development approaches and explained how changes to related artifacts act as the trigger for traceability maintenance. This means that, in order to enable traceability maintenance in a (semi-) automated way, it is necessary to recognize these changes to related artifacts, and then to determine and perform the required update to the impacted traceability relations. Our approach focuses on maintaining traceability as a by-product of changes made to structural UML models during object-oriented software development. Our work is based on version 2.3 of the UML, which was the latest version at the time of this work. This leads to two natural phases in the approach:

Phase 1 Recognition: Capturing elementary changes to model elements and recognizing the compound development activity applied to the model element, as comprised several elementary changes; and
Phase 2 Maintenance: Updating the traceability relations associated with the changed model element.

Fig. 4 illustrates these two phases using an example that replaces a class within a design model, described as a UML class diagram, with a component. The development activity consists of three elementary changes: deleting class *Printer*, creating a new component and renaming the *New Component* as *Printer*. The required traceability update is re-creating the two traceability relations that existed on class *Printer* on the component *Printer* after the activity has been completed.

3.1.1. Development activity recognition

Relevant changes that might require the maintenance of traceability are those that alter artifacts of related models (Maletic et al., 2005). Such changes comprise a sequence of one or more incremental changes (Cleland-Huang et al., 2002). This paper refers to these as elementary changes.

While identifying elementary changes to artifacts is mostly a technical problem, the recognition of compound change activities consisting of multiple elementary changes can be a complex task with high uncertainty. This paper refers to such sequences of elementary changes as development activities.

A decision that had to be made during the development of the approach was whether to perform the development activity recognition process automatically or with user support. This decision depends on the required certainty in the change recognition, the acceptable manual effort and the influence permitted on the working process of the user. Models described in a semi-formal language

support different types of elements and follow a general definition within a meta-model. An idea that emerged during the development of the approach was to use this meta-information about a model in order to omit user support for the recognition process. Additional information about changing elements (e.g., type, name, type of the parent element) and knowledge about possible, allowed and meaningful changes to the different types of model element allow for the identification of a limited set of development activities.

To demonstrate and study the approach in depth, it was necessary to decide which type of model to support. We focused on structural UML diagrams as most practitioners interviewed about their traceability practice referred to related structural UML diagrams within their software development process (Mäder et al., 2009c). In particular, the development activities considered so far involve the following model elements: class, component, package, attribute, method, association, dependency, inheritance and stereotypes of these (e.g., aggregation, composition, association class and interface). Accordingly, all diagrams containing these element types are supported by our approach.

3.1.2. Traceability relation maintenance

Following the recognition of changes, the subsequent maintenance of traceability relations is based upon two premises. First, given a model element related by traceability relations, along with knowledge of the element(s) that replace the initial one after a development activity, all traceability relations of the initial element should be present also on the evolved element(s). Second, given a model element related by traceability relations, along with knowledge about its modification (moving one of its parts into another artifact), those traceability relations on the element related with the moved part should be copied or moved to the other element. Those traceability relations related only to the part should be moved, while those also related to remaining parts of the original element should be copied.

In this paper, the elements that are involved in the update of traceability relations are distinguished into update sources and targets. The approach discussed in this paper is called (semi-) automated as, in many cases, it is not possible to determine, which traceability relations of an element refer to which of its parts, unless explicitly specified. The occasional need for manual intervention is therefore discussed throughout this paper.

3.2. Assumptions

The approach is based on the following assumptions:

- Model-based development of a system using UML and SysML diagrams for modeling the structure of the system (e.g., for analysis, design or implementation models). The semi-formal nature of both UML and SysML models supports the recognition of changes, and their use is common in industry (Mäder et al., 2009c). The focus on these kinds of models means that the approach is concerned solely with maintaining post-requirements traceability (Gotel and Finkelstein, 1994).
- The existence of an up-to-date traceability information model, defining permitted traceability relations for the project (Mäder et al., 2009d). A traceability information model (TIM) is a graph defining the permissible trace artifact types, the permissible trace link types and the permissible trace relationships on a project, in order to address the anticipated traceability-related queries and traceability-enabled activities and tasks. In our context, the TIM allows for automated traceability updates in accordance to the traceability strategy of a project.
- A pre-existing set of traceability relations established between model elements in accordance to the traceability information model and stored within a traceability relation repository. A traceability relation repository provides database-like features for storing, changing, and querying traceability relations.
- Only a limited number of development activities evolving a model element are performed in parallel by one developer. It is assumed that a developer will finish her/his development activities with only a limited number of intermediate elementary changes belonging to other development activities. The concrete value of acceptable parallel development activities remains configurable by the user. Issues arising from multiple users performing concurrent changes to model are not within our scope as these should be handled by the modeling tool.
- Changes to related models are undertaken within a Computer Aided Software Engineering (CASE) tool. The use of a CASE tool eases gaining access to change information.

Given these assumptions, the approach supports the following scenarios: (I) the change of a model within the same level of abstraction, typically to evolve the model as a result of changing or new requirements (e.g., evolving the analysis model); and (II) the change of a model into a more abstract or detailed level of abstraction, typically to explore requirements realization (e.g., refining the analysis model into the design model). There are no restrictions as to those artifacts that can be related, but the approach will maintain only those ends of traceability relations that reside on an element that is part of a model expressed as a structural UML diagram.

3.3. Challenges and concepts

The main challenge related to Phase 1 of the approach is identifying development activities within a flow of elementary changes. This includes:

R.1: Relating several elementary changes to one development activity. The type of an elementary change and that of the impacted model element do not offer enough information to relate elementary changes to one another. It is necessary to compare additional properties of the changed element, like its identity through several changes or the type of the element containing the changed one.
R.2: Recognizing different sequences of elementary changes (i.e., different ways to perform an activity) as the same development activity.
R.3: Recognizing different orders of the same elementary changes as the same development activity.

The main challenge related to Phase 2 of the approach is identifying impacted or missing traceability relations and updating them. This includes:

M.1: Defining those elements of a development activity that hold the initial relations (update sources) and those elements that receive these traceability relations after the development activity (update targets).
M.2: Determining those traceability relations of a modified element that are impacted by moving one of its parts to another element.

To address these challenges, the approach uses traceability maintenance rules to define development activities to be recognized and traceability updates that have to be carried out. The use

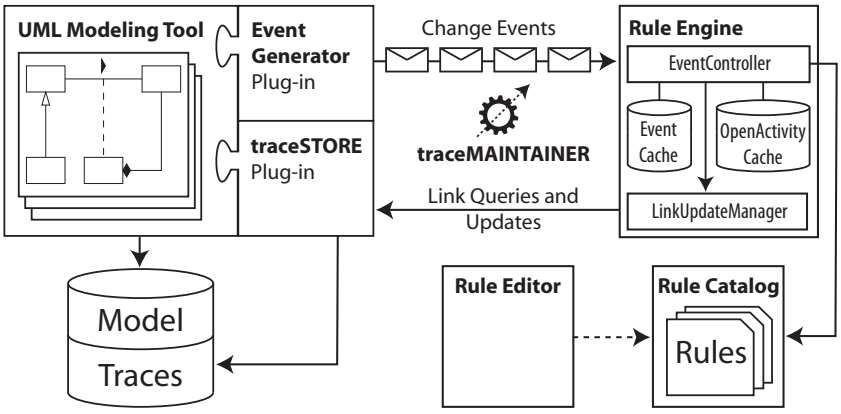


Fig. 5. Technical overview of the approach

of rules is not new in the field of traceability (see Section 2.2.2), but in contrast to approaches that apply information retrieval techniques to identify traceability relations between two sets of artifacts, the introduced rules allow the user to support and customize the recognition process with additional properties of searched relations (e.g., information about the structure of the artifacts).

Traceability maintenance rules allow to define abstract and valid sequences of elementary changes that establish the same development activity. A rule consists of two parts, one that defines the development activity to be recognized and one that specifies the traceability update to be carried out. Since most development activities can be performed in multiple ways, in terms of underlying elementary changes, a rule consists of alternative sections grouping the definition of one specific way to perform the activity and the correlating traceability update directives, addressing Challenge R.2. That part of an alternative section that defines the development activity is called the change sequence and defines elementary changes that are necessary in order to perform the development activity in that alternative way. Abstract elementary changes are called masks, a concept that addresses Challenge R.1. The update part of a change sequence allows update sources and update targets to be specified, addressing Challenge M.1. All concepts used by our rules will be explained within the next section, and the reader can find an example rule in Listing 1.

The defined rules are compared with captured elementary changes and, once a match has been detected, the defined traceability update within the rule is carried out. A match can only be detected if all the elementary changes have been performed and can be evaluated. For that purpose, a buffer holds a number of recently performed changes for later comparison. This buffer, along with the concept of masks and property references (described more fully in Section 4.3), allows a development activity to be recognized in any valid order that the elementary changes can be performed in, addressing Challenge R.3. The traceability update is performed once a development activity has been completely recognized. If an element is modified during the activity, by moving one or more of its parts to another element, then the user is asked to highlight the impacted traceability relations. This addresses Challenge M.2.

3.4. Technical overview

Fig. 5 gives a technical overview of the approach, setting the discussed concepts in relation to each other. The left side of the figure depicts a development tool holding models and traces for a given project. That tool is being extended by two plug-ins. The first plug-in, Event Generator, recognizes changes to development models, captures them as change events and transmits them to the Rule Engine for analysis. The second plug-in, traceSTORE, gives access to the traceability within the development project and is able to carry out determined update actions once a development activity has been recognized and requires traceability maintenance. The right hand side of Fig. 5 depicts the developed Rule Engine and the Rule Catalog as well as its main concepts, which will all be discussed in depth within the next two sections. A developer performing manual changes to the depicted model will be observed by capturing the elementary changes made to the model as change events. As soon as a number of change events are recognized as being one of the defined development activities within the Rule Catalog, and where traceability has become outdated due to the performed changes, the approach automatically or semi-automatically updates the impacted traceability.

The approach to development activity recognition is discussed in depth in Section 4, while the update process for traceability maintenance is discussed in depth in Section 5.

4. Development activity recognition

This section describes Phase 1 of the approach and its underlying concepts.

4.1. Change events

Section 3.1.1 introduced elementary changes to model elements. We assume the use of a CASE tool that allows for the capture of such elementary changes and that issues change events containing the captured information (e.g., Sparx Enterprise Architect). A change event is issued if an elementary change altered at least one property of interest of an element.

There are three fundamental types of change to elements: add, delete and modify. In addition to the type of change, information is captured about the properties of the model element that the change is applied to (e.g., name and identifier), as property value pairs. For the addition of an element, these properties only exist after the creation of the element, while for deletion they only exist before destruction. For the modification of an element, both pre and post modification properties are required for analysis. Three change events are therefore distinguished: *ADD*, *DEL* and a composite *preMOD*|*postMOD* event.

In principle, change events to all supported element types of a modeling tool could be issued and they could capture all available properties of the elements. Such an approach would cause for more complex models an enormous number of events, potentially slowing down any processing with data that is rarely used. For that reason, an event configuration represented as a class diagram defines the element types of interest along with the properties that are needed to recognize defined development activities (see Mäder et al., 2008b for further information). This diagram can be enhanced iteratively if additional element types need to be supported or if additional properties are needed. The minimal required properties of an element are its identifier and its type; they have to be invariable over the life of an element and thus enable unique addressing. Properties can also reflect the state of another element related to a changed one (e.g., parent, end1, end2, dependent, independent, sub and super). A parent element is available for all elements of a UML model due to the hierarchical order of these models. Other related elements are specific to certain element types (e.g., end1 and end2 are only available for associations within UML models).

4.2. Development activities

The success of the approach depends on the ability to capture traceability relevant changes to related model elements. It is not sufficient to separately examine the change events discussed before as these reflect only a single change to a model element, not necessarily the whole transformation of an element into one or more evolved elements, requiring traceability updates. Such transformations are identified by examining several change events in relation to each other.

Development activities are changes that happen and recur, for example, while developing the design of a system, refining an abstract model into a more concrete model, and during the corrective and evolutionary maintenance of systems. Of interest are all UML classifiers and relations that establish the structure of a system. While evaluating possible changes to elements of these types, six basic categories of development activity have been recognized that require traceability update:

1. Adding an element.
2. Deleting an element.
3. Replacing an element.
4. Merging several elements into one whole.
5. Splitting an element into parts.
6. Modifying an element by adding or removing parts.

These development activity types correlate with those that have been identified by Cleland-Huang et al. (2002) for the evolution of requirements artifacts (see Section 2.2.1). These activity types are discussed along with the necessary traceability update in Section 5. In this current section, the focus is on recognizing compound development activities from a flow of elementary changes. This observation has also been made by Cleland-Huang et al. (2002), where the authors decided to facilitate the recognition process by user support.

4.2.1. Identifying development activities

To define possible development activities to structural UML models, one could start to generate all possible permutations between types of model elements and the categories of development activities listed before. The disadvantage of such an approach is that it would generate a large number of development activities, many incorrect in relation to the UML meta-model or with no semantic meaning (e.g., replacing an attribute by a method or splitting a class into two associations).

A different approach has been chosen to define a comprehensive list of possible and meaningful development activities. Several development methodologies, as well as industrial projects, were studied and traceability relevant change activities that usually occur during the analysis and design of systems, or due to evolutionary changes, were collected. Forward engineering processes that were studied include the Unified Process (Jacobson et al., 1999; Kruchten, 2000; Arlow and Neustadt, 2005), Fusion (Coleman, 1994) and Quasar (Russek, 2004; Siedersleben, 2004). Refactoring activities (Fowler, 1999) were also studied and included in the list of activities. Among all the refactorings suggested by Fowler, those of interest were those that altered the structure of a development and could possibly cause the need for traceability maintenance (e.g., Move Class). We found that most of the relevant refactorings were already covered by existing development activities included for the forward engineering processes. In addition, the discussion of systems design with UML of Lano (2005) provided additional candidates for development activities. As a result of these studies, the current rule catalog for structural UML models comprises 38 development activities (13 apply to associations, 4 to inheritance, 4 to attributes, 2 to methods, 5 to classes, 6 to components and 4 to packages) defined as 19 rules with 67 alternatives (Mäder, 2009). The catalog has been improved multiple times and has been used in its current form during several studies and experiments (Mäder et al., 2008a,b, 2009a), one of them discussed in Section 6. A few example development activities, along with their type according to the categories introduced before, are listed in the following:

Development activity examples applied to relations:

- Refining an unspecified association into one directed association (type: replace)
- Refining a bidirectional association into two unidirectional associations (type: split)
- Refining an association into aggregation or composition (type: replace)
- Resolving a one to many association (type: split)
- Resolving a many to many association (type: split)
- Resolving an association class (type: split)
- etc.

Development activity examples applied to classifiers:

- Moving attribute, method, class, component, package (type: modify)
- Splitting class, component, package (type: split)
- Merging class, component, package (type: merge)
- Converting class into component (type: replace)
- Converting attribute into class (type: replace)
- etc.

Several development activities are captured by more than one rule, e.g., splitting a class, package and component. Splitting and merging of elements are recognized by a move of their parts, e.g., the move of attributes and methods between classes for recognizing class merge and split. Some of the activities are only traceability relevant if the impacted model element is being deleted and a new element created, instead of modifying the existent one (e.g., refining an association to an aggregation), because only in the former case do existing traceability relations become disconnected and have to be recreated on the replacing element. This is the reason for the difference in the number of activities and rules. We published a list of all identified development activities and a list of all defined rules in the appendices of Mäder (2009). In this document, we provide an explanation for each development activity and relate it to the covering rules.

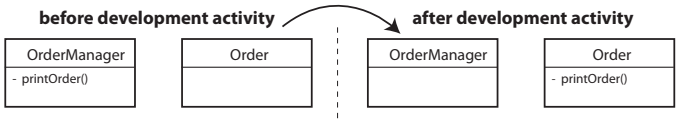


Fig. 6. Example activity: moving a method between two classes

The obtained list of development activities is unlikely to reflect all the development activities of interest and applicable to structural UML models, but it has been found sufficient to study the approach and provided encouraging results during evaluation. The idea is to provide an initial and stabilized rule catalog to the users that recognizes common development activities. If needed, further customization of existing rules and the definition of new ones can be done by a trained user. The strategy for improving the rule catalog is iterative:

(I) Identify a development activity to a model element that is not currently supported by the existing rule catalog, but requires traceability maintenance.
(II) Find all different sequences of elementary changes, in terms of change type and impacted element type, that can be performed within the supported modeling tool to execute the activity in (I).
(III) Define each discovered way to perform the activity and compose all alternative ways into one new rule.
(IV) Define the necessary update to traceability relations and add descriptive information to the rule (discussed in Section 5).

If one of the changes that take part in the development activity is applied to an element of a type for which change events are not yet generated, or if not all required properties are captured in the currently issued events, then it is necessary to define that element type or required properties in the event configuration (described in Section 4.4.1).

4.2.2. Sequences of elementary changes

The aim is to recognize development activities in a flow of change events triggered by a developer working on a model. This requires knowledge about valid sequences of change events to perform an activity. This information can be gathered by finding out about the possibilities a CASE tool provides to change model elements. The following example illustrates that process with a simple example (see Fig. 6). The left part of the figure shows the model before the development activity and the right part shows it after completion.

The method *printOrder* has been moved from class *OrderManager* to class *Order*. Such an activity can become necessary due to a shift in the responsibilities of both classes. A modeling tool allows the user to perform the activity in two alternative ways that cause different change events:

(I) Move the method by drag and drop. By performing the activity in this way, the initial method is preserved. The move between both classes can be identified by comparing the *preMOD* and *postMOD* event generated during the change.
(II) Deleting the initial method and adding a similarly named one. By performing the activity in this way, the method is being deleted (*DEL* event) and then recreated (*ADD* event).

4.3. Abstract development activities

In theory, a solution to recognize development activities could be defining exactly each activity that is intended to be recognized but, in practice, the large number of different ways to perform an activity makes that approach impracticable. It is necessary to provide concepts that allow for those properties of a flow of elementary changes that are not specific to the development activity to be delineated, while still relying on those properties that characterize the activity and distinguish it from others. An abstract development activity is one that is compliant with all valid ways to perform the activity, but not compliant with any other. It is represented as a rule and incorporates the following concepts:

- Masks – to allow a group of elementary changes with the same characteristic properties to be defined.
- Property references – to allow abstracting from concrete property values by defining dependencies between property values of two events that have to have related.
- The *EventCache* – to allow abstracting from concrete orders of elementary changes by providing a history of recent incoming events in order to compare events once enough information becomes available.
- Alternatives – to allow the grouping of different sequences of elementary changes constituting the same development activity.

While we were not able to simply reuse a whole existing notation, abstract development activities and traceability maintenance rules incorporate several existing concepts. We considered graph transformation rules as notation, but these are intended to match a given, static state of a graph, while we are looking for a sequence of changes to a graph. We do not claim that there is no other notation that could be modified to our purpose, but we were not able to find an out of the box solution that could simply be applied.

4.3.1. Masks

A development activity consists of a number of elementary changes. These changes have a distinct type (i.e., add, delete or modify) and will be applied to a certain type of model element (see Section 4.1). Furthermore, each change event provides a number of additional properties that describe the changed element before and/or after the change. Some of these properties are characteristic for a development activity (e.g., for a certain activity the name of an element has to stay the same before and after the activity), but usually not all properties are relevant (e.g., the stereotypes that are attached to a moved element may have no relevance). In order to recognize a development activity, one wants to compare incoming events generated due to model changes of the user with predefined events belonging to known development activities. To prevent the necessity to define each concrete event that could comprise a certain step of a development activity, masks provide a way to define only the characteristic properties and so a whole set of matching events. A mask defines those properties of a matching change event that have to take certain values, while ignoring those that may take any value. The construct is called mask as it has some similarities with subnet masks used in the networking domain (see for example the RFC 950 standard). One can imagine laying the mask over incoming events in order to compare the properties defined within the mask with those of the incoming event. The evaluation of all single property comparisons provides a boolean result, whether event and mask are matching or not.

Values of the properties can be defined as static expected values or as references to the properties of another mask within

the same development activity (see Section 4.3.2). Two properties of an expected event have to be defined for each mask, the change type and the element type. This restriction results from the necessity to find a compromise between abstraction from concrete events in order to save effort for the definition of rules and to keep the defined rules easily comprehensible by humans to allow their customization. The following shows an example of a mask that matches with incoming change events indicating the deletion of a method named *printOrder* from a class *OrderManager*:

```
DEL(type='method'; id=*; name='printOrder';
    parent.type='class'; parent.id=*;
    parent.name='OrderManager')
```

The * symbol is a wildcard that defines a property that can have any value within an incoming event while still being compliant with the mask. In the example, the id's of the method and that of the enclosing class as well as applied stereotypes to the method may take any concrete value in a compliant event. To be able to allow multiple valid values for one property, or to exclude certain values, boolean logic is supported for the definition of property values. By using boolean logic, it is possible to allow any but one value for a property (e.g., type=¡class') or to allow for multiple values (e.g., stereotype='use' || 'realize').

4.3.2. Property references

Masks allow characteristic properties of an elementary change to be defined. In most cases, this is not sufficient as a concrete expected value is not known at the time of rule definition. What is known is the relation between elementary changes, for example, that a property of one elementary change has to have or must not have the same value as a property of another elementary change that participates in the same development activity (e.g., the name of the replacing element must be different from that being replaced).

The approach addresses this problem by allowing the definition of the value within one mask as a reference to a value of a property within another mask. This means that the value is expected to be the same as or distinct from the value of the referenced property of a change event that will be assigned to a different mask of the same development activity. These references between masks allow elementary changes to be related to one another. References consist of an identifier of the mask and the name of the property within that mask that they refer to. In order to be able to resolve references, circular references between two or more masks must be prevented.

The following example shows two masks with references between them. These masks work together to help match a rule. An event that matches mask 1 would be any that indicates that a method has been deleted from a class. An event that matches mask 2 would be any that informs about the addition of a method with the same name as the method that has been assigned to mask 1 (mask 2: name=1.name). Furthermore, mask 2 requires that the method has not been added to the same class that the one assigned to mask 1 has been deleted from (mask 2: parent.id=!1.parent.id).

```
  (mask 1) DEL(type='method'; id=*; name=*; par-
ent.id=*;
      parent.type='class'; parent.name=*)
  (mask 2) ADD(type='method'; id=*; name=1.name;
      parent.id=!1.parent.id; parent.name=*;
      parent.type='class')
```

4.3.3. The EventCache

Identifying a development activity requires comparing at least two states of a model, before and after a change. This means that at least two masks have to be defined as a change sequence to recognize a compound development activity. Such change sequences can often be carried out via various orders of underlying elementary changes. Referring back to the example of moving a method, there is no difference in terms of the overarching applied development activity if the method is deleted first from the initial class and added afterwards to the new class or vice versa. The number of possible permutations is defined by the number of elementary changes contributing to a development activity and is constrained by the existence of elements (e.g., an element cannot be modified or deleted before it has been created and the corresponding events in that case are not interchangeable). However, the order in which a development activity is performed has no influence on the characteristic properties of the triggered change events, which means that they do not vary depending on the order in which a development activity is performed.

This enables only one sequence of masks to be defined, independent of the order in which events belonging to an activity arrive. The order of the masks within the change sequence does not imply any required order of the incoming events. Nevertheless, most masks will be defined as dependent upon other masks by referencing values of their properties, which means that it might not be possible to compare a matching change event immediately after its arrival if the mask in question has references to another mask that has not yet been assigned to an incoming event. To address this situation, a number of past incoming events is held in an *EventCache*. The concrete process of comparing events with masks has been discussed in Mäder et al. (2008b).

Having a set of past incoming events and a number of abstract activity descriptions (defined as a set of masks) raises questions about when and how to start comparing them. As discussed before, references that one mask have to another can only be resolved if an event has already been assigned to the referenced mask. To start that mechanism, one mask within each change sequence is required that must have no references. This mask is called the *TriggerMask* and the assignment of an event to that mask triggers the recognition process.

4.3.4. Alternatives

To address the issue of performing a development activity in multiple ways, several change sequences can be grouped as one rule that is able to recognize the same overarching development activity. The different change sequences performing the same activity are called alternatives. Alternatives can be seen as a grouping of similar change sequences transforming an initial state of model elements into the same final state.

4.4. Rules

All the concepts discussed in Section 4.3 have been incorporated in the structure of rules and comprise a rule catalog. The rules recognize development activities and hold information about the necessary traceability update after recognition.

As an example of a concrete rule, Listing 1 shows the rule to recognize the move of a method between two classes. The rule definition contains a name, description, and two alternative ways to perform the activity. The description is presented to the user upon recognition of the activity, in cases where user interaction is required. Numbers preceded by a percentage sign are placeholders containing concrete model element information at runtime. Alternative 1 requires to delete the method from one class (mask T) and to add it to another class (mask 1). The characteristic properties of this change sequence are defined with mask 1, the added method has to have the same name as the deleted one, it may not have the same id as the deleted one and may not be added to the class it has been deleted from. Alternative 2 requires to drag the method from one class (mask T) and to drop it on another class (mask 1). The

characteristic properties are also defined with mask 1, the dropped method has to have the same id and may not be dropped over the class it has been dragged from. It is a rather simple rule, but shows all the main concepts.

Listing 1: Rule to identify the moving of a method between two classes

```
<Rule id='9'>
  <Name>Move method</Name>
  <Description> %1 %2 was moved from %3 %4 to
    %5 %6.</Description>
  <Alternative id='1'>
    <ChangeSequence>
      <Mask id='T' type='DEL'>
        <Element type='method'/>
      </Mask>
      <Mask id='1' type='ADD'>
        <Element id=!T.id type='method'
          name=T.name parent.id=!T.parent.id/>
      </Mask>
    </ChangeSequence>
    <LinkUpdate> <!- see Section 5.2 ->
      <UpdateSource relationsOn='both'>
        T.id</UpdateSource>
      <UpdateTarget relationsOn='both'>
        1.id</UpdateTarget>
    </LinkUpdate>
    <DescriptionPlaceholders>
      <Placeholder id='1'>T.ElementType
        </Placeholder>
      ...
    </DescriptionPlaceholders>
  </Alternative>
  <Alternative id='2'>
    <ChangeSequence>
      <Mask id='T' type='preMOD'>
        <Element type='method'/>
      </Mask>
      <Mask id='1' type='postMOD'>
        <Element id=T.id type='method'
          parent.id=!T.parent.id/>
      </Mask>
    </ChangeSequence>
    <LinkUpdate> <!- see Section 5.2 ->
      ...
    </LinkUpdate>
    <DescriptionPlaceholders>
      ...
    </DescriptionPlaceholders>
  </Alternative>
</Rule>
```

4.4.1. Rule definition

The traceability maintenance rules are stored in the open XML format. The definition of the rule catalog's structure and of the included rules is distributed between an XML Schema Definition (XSD) and the event configuration that defines change events (see Section 4.1). The schema describes the general structure of the rule catalog and of all those elements of the catalog that are independent of the supported model. The event configuration describes those parts of the catalog that are dependent upon the supported model elements, that is, the element types and respective properties that can be used for defining masks. Separating the catalog's structure definition into an XML schema and event configuration makes the approach and prototype implementation independent of a certain model and allows the user to customize the support for model elements. Further information can be found in Mäder et al. (2008b).

The definition of rules can be challenging. For that reason, a rule editor was developed to assist with rule creation, editing and validation (see Mäder et al., 2009b). The editor validates rules against the current catalog and event definition and performs automated checks to address common problems when evolving a rule catalog, such as:

(I) Structural issues – inconsistencies within the structure of the rule catalog can be identified.
(II) Element type and property inconsistencies – unsupported element types and properties of elements that are not part of the event configuration can be found.
(III) Syntax errors within property values – the definition of property values can be validated.
(IV) Reference specification errors – the existence of referenced masks and properties can be checked.
(V) Reference dependency errors – to make a rule satisfiable, one mask without references (i.e., the *TriggerMask*) is necessary, along with an overall tree-like reference structure starting from this mask, with no cyclic dependencies. This can be checked.
(VI) Inclusion of rules – inclusions of one rule within the context of another can be identified.

The editor further allows the update to the traceability relations to be performed on the recognition of a development activity to be specified. It also allows for any user notifications to be defined where interaction is required.

4.4.2. 'Good' rules

A 'good' rule is one that is distinct, complete and satisfiable. It is important to define a rule that distinguishes itself from other rules. Therefore, it is necessary to find all those properties that are characteristic for the development activity. It is also important to understand and define all the ways a user may perform a given development activity, transforming an initial model state into the same final state, and capture all these ways within the rule.

The required properties comprising a mask can be defined as boolean expressions, as discussed in Section 4.3.1. It is important that these expressions are satisfiable (the SAT problem; Du et al., 1997). Furthermore, the assignment of an event to a mask requires resolving all references within the mask in order to be able to compare it with incoming change events. These references form a graph over all related masks of a change sequence. To make this structure resolvable and satisfiable it is necessary to have a non-cyclic tree-like reference structure starting from a *TriggerMask* that has no references.

One development activity might be completely part of another (e.g., an unspecified association could be replaced by one directed association or by two directed associations). In such cases, it is not possible to define two distinct rules that permit one case to be recognized without the other. The solution is to have a rule for the partial activity to guarantee that an action is performed, but also a rule for the larger composite activity. To prevent false traceability updates, it is only necessary during rule definition to keep in mind that the partial rule might already be fired and the interim action completed before the further rule is fired and the larger activity completed. The rule editor discussed in Section 4.4.1 provides a function to automatically identify inclusions and to support the user in the rule definition.

A change sequence within a rule defines expected changes in order to recognize the overarching development activity. Obvious questions are: what happens if the developer performs expected changes to a model element as multiple, incremental changes; or what happens if the developer is correcting a mistake? To answer these questions, it is important to note that a mask does not define an expected concrete change increment to an element, but a change type and an expected state of the element afterwards, in terms of its properties. This means that for multiple, incremental changes and detours, an event is generated every time the developer commits a change, but these events do not match with the mask defined

within the rule and so will be ignored until the event comes in that shows the element in the expected state.

4.4.3. Rule application

The task of matching elementary changes with traceability maintenance rules requires an effective rule engine. Such reactive systems are built according to the Event Condition Action paradigm (ECA) (van Bemmel et al., 2004). This paradigm defines systems that trigger an action after an incoming event has matched a defined condition. As this approach is not intended to react to a single event, but to react to patterns over the event history, a much more sophisticated rule engine with Complex Event Processing (CEP) (Luckham, 2002) was necessary. This has been developed as part of the prototype system that implements the approach (see Mäder et al., 2008b, 2009b).

4.5. Critique of Phase 1

The quality of the proposed approach to development activity recognition depends upon two aspects: (I) the completeness of the rule catalog in terms of defined development activities and the ways in which they could be achieved; and (II) the quality of the defined rules, so the degree to which a rule correlates with the underlying development activitiy. Both aspects are discussed in the following sections.

4.5.1. Expression power of rules

In this section we discuss the limitations of our technique in terms of expression power. Our technique relies on the semi-formal nature of the analyzed structural UML diagrams. That means known model element types and their properties, defined within the UML meta-model. That information is used to relate elementary changes to each other. Our masks allow using all these properties of model elements to definite development activities. Limitation of our technology arises from whether the available properties are sufficient to define a cohesive activity that can certainly be recognized and distinguished from others.

We can state that the expressiveness was sufficient to define all the discovered development activities during our initial study discussed in Section 4.2.1. Nonetheless, there are activities that cannot be recognized with sufficient certainty. As example, we are discussing the splitting and merging of methods that are not currently covered by our rule catalog. Methods are elementary, non-dividable entities within a structural UML model and as such do not offer sufficient properties for recognizing their splitting or merging automatically. As opposed to the splitting and merging of compound entities, e.g., classes, components, and packages, which is recognized by a move of child elements, e.g., methods, attributes, and classes.

This limitation is acceptable as these are fine-grained activities, usually performed on the source code level and not within analysis and design models. Our approach supports activities at the same level of granularity as the notation does. If we would extend our approach to development activities in source code, we could recognize the splitting and merging of methods by observing moves of source code parts between methods and so recognize the splitting and merging of methods.

4.5.2. Completeness of the rule catalog

Through experimentation, the current rule catalog appears to be stable (as discussed in Section 6). However, it is not possible to ensure the completeness of all development activities in the catalog due to the semi-formal nature of UML models. The current catalog can provide a common set of rules for the supported model, as the evaluation will show, but it is likely that it has to be further enhanced and improved. It can also be necessary to customize single rules to the specific needs of a developer (e.g., to support different ways of refining elements). In this context, an incomplete rule catalog means fewer traceability relations are maintained in an automated manner than could be.

4.5.3. Quality of the rule catalog

The quality and correctness of the defined rules are very important for the success of the proposed approach and will get even more important as the catalog expands. Issues that may arise can be classified into two generic categories:

(I) False recognition – a rule fires in situations it is not intended for.
(II) Missing recognition – a rule does not fire in a situation where it should.

Both problems can be caused by an issue that could be referred to as finding the right abstraction between rule and development activity. If a rule is too abstract and does not constrain properties sufficiently, then it will fire in those situations where the sought development activity has not taken place, potentially leading to false changes to traceability relations. If properties are constrained too much, a rule will not recognize the development activity in all the ways it can be performed, potentially leading to missing changes to traceability relations. These issues cannot be found by analyzing the rule catalog, but they can be examined by performing experiments and black-box tests with a given catalog. A missing recognition can also be the result of syntax or reference failures within the rule specification. A missing recognition might also happen due to unsatisfiable boolean expressions in the property definition or due to unresolvable, circular dependencies between masks. Both these issues are addressed by validation tests within the rule editor.

5. Traceability relation maintenance

This section describes Phase 2 of the approach and its underlying concepts.

5.1. Development activity types

Focusing on post-requirements traceability relations, each related requirement spans a graph towards implementation artifacts (e.g., analysis elements, design elements and test cases). The nodes of this graph represent related elements in the same or different models, and the arcs represent traceability relations between these elements. Each related requirement spans such a tree. Fig. 7 depicts an example graph of two requirements and their related implementing artifacts (red and green highlighted). The graph also depicts possible changes to related elements and their impact on existing traceability relations (gray shaded). The directionality of the arcs expresses dependence between related artifacts (see Section 2.1). Correlating to the development activity types discussed in Section 4.2, the following changes and associated traceability updates to the graph are possible and depicted in the figure:

- A new node can be added to the graph. Traceability update: create trace(s) on new element.
- An existing node can be deleted from the graph. Traceability update: remove trace(s) connected to deleted element.
- An existing node can be replaced by another node. Traceability update: restoration of all traceability relations of the replaced model element on the replacing element.

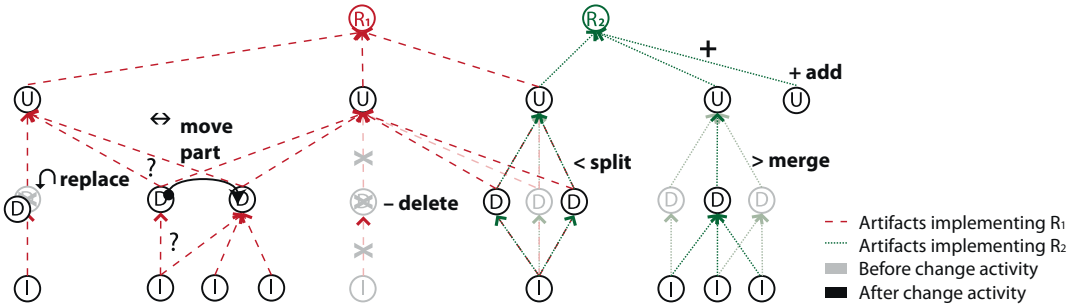


Fig. 7. Example of two overlapping traceability graphs implementing two requirements (R_1 and R_2), motivating different possible development activity types. The nodes of the graph refer to the following element types: R – requirement, U – use case, D – design element and I – implementation element.

- An existing node can be split into two or more nodes. Traceability update: copy traceability relations to all resulting elements of the split activity.
- Two or more existing nodes can be merged into one node. Traceability update: combine the traceability relations of all merged elements on the resulting element.

In addition to the changes before, consisting of adding and/or deleting nodes, it is possible to change existing nodes. This means to add or to delete sub-elements to nodes. These changes can have an impact on the traceability graph as well. Fig. 7 depicts the moving of a sub-element between two nodes and the possible impact on the traceability relations of the containing nodes.

Structural UML models are hierarchical meaning that, except for the root package, each other model element is part of an enclosing parent element. Therefore, the discussed change activities always happen inside one or more enclosing parent elements (e.g., a class is being added to a package or a method is being deleted from a class). Moving one step higher in the hierarchy and looking at the parent elements, each addition of a new element and deletion of an existing element is also a modification of the enclosing parent element. Each of the development activity types discussed so far may move an element into a different parent element. Examples would be moving a class into another package or splitting a class into two classes residing in different packages. Changing the context of an element may require copying or moving the traceability relations on the element's parent to the new parent.

In cases where more than one traceability relation reside on the source parent element(s), it is not possible to automatically determine which relation is impacted by a move and a decision from the user is required. A dialog is provided to the user that shows all the traceability relations on the former parent element (old context), along with options to delete them from the old parent (update source) and to create them on the new parent (update target). This approach provides the developer with the possibility of either leave, copy or move for each traceability relation.

5.2. Update specification and execution

To enable the approach to perform traceability updates automatically, it is necessary to provide update directives for each recognized development activity and to relate them to concrete model elements that hold the traceability relations. The process of specifying and executing the traceability updates is described in the following.

5.2.1. Specifying the update

A rule consists of mask sequences, representing different ways to perform a development activity and information that specifies how to perform a traceability update once the defined activity has been recognized. This update information is not only specific to a certain development activity, but also to the concrete change sequence that has been recognized. This means that it is specific to the different alternatives within a rule (see Section 4.3.4).

Update sources and targets are defined as property references to element ids within masks for each alternative of an activity. The *id* identifies a model element and allows the retrieval of traceability relations from the repository during the update procedure. For each update source and each update target, an additional attribute *relationsOn* is defined. The attribute can have one of the following three values: *'element'*, *'parent'* or *'both'*. For an update source, the attribute specifies which traceability relations to take into account for an update, those of the defined source, those of its parent element or those of both. For an update target, the attribute specifies where to place the new traceability relations, on the target element itself, on its parent element or on both.

During the update procedure, the relations of all providing elements to all receiving elements are propagated. An exception is the combination where update source and target both have their attribute *relationsOn* defined as *'both'*. In that case, the relations on the update source element will be propagated to the update target element and those of the update source's parent element will be propagated to the update target's parent element.

5.2.2. Executing the update

Once a development activity has been recognized, the rule engine carries out the following steps to execute the necessary update:

Step 1: Retrieve Impacted Relations – All the traceability relations of all update source elements and/or their parent elements, as well as those of all update target elements and/or their parent elements, are retrieved (see Section 5.2.1). All these relations are added to an update list which eventually provides information about all the necessary changes to traceability relations. If no traceability relations exist on any of the update source elements, then no update is required and the procedure ends immediately.

Step 2: Add New Relations to the Update List – Depending on whether there are traceability relations on at least one of the update sources, the update list is extended with potential new relations. The underlying concept is to copy traceability relations that exist on the update source elements to the update target elements, if not already existing. For that process, each update source is compared pair-wise in terms of existing relations with each update target, and not yet existing relations are added to the list as proposed.

Step 3: Define an Update Action for Each Relation – The required update action for each relation in the update list is determined. Possible actions for proposed (i.e., not yet existing) relations are to create or discard (no creation), and for currently existing relations

are to stay (no change) or delete. The required update action is determined according to the following directives:

D1 The update action for relations on the update source(s) is defined to stay if the source element still exists. It is defined to delete if the element has been removed during the development activity.
D2 For relations on the parent element of update source(s), the update action cannot be determined automatically in most cases and the user is required to decide between stay and delete for each relation. Optionally, for situations with exactly one relation on the parent element, the action for this relation can be defined to stay automatically.
D3 Proposed relations on the update targets are defined to be created.
D4 For proposed relations on the parent element of update targets, no distinct action can be defined automatically and the user has to decide between create and discard. Optionally, if only one relation was existing on the parent element of the update source, then this relation can be defined to be created automatically.
D5 All existing relations of all update target(s) and its parent elements are defined to stay. These relations are not impacted by the development activity, but they are included in the update list to provide the whole context of an update during user interaction.

Step 4: Execute the Update – In this step, all the actions defined for the traceability relations in the update list are transformed into commands for the traceability relation repository. The relation repository processes these commands and completes the update.

5.2.3. Project-specific traceability maintenance

An issue that has not been discussed so far is the customization of updates according to the intended traceability for a project, as defined in the traceability information model. The specified catalog includes rules to recognize generic development activities with possible impact on traceability relations for a supported model. Furthermore, the updates within each rule are defined in a general way to update impacted traceability relations on all update source elements participating in the development activity. The idea is to have a general catalog not requiring customization for each new project in order to reduce the effort of using the approach.

A problem that could arise is that traceability relations are updated or created on elements that are not intended to be traced for a given project. Even if a rule exists within the catalog and is triggered by a development activity of the developer, an update is only performed if traceability relations exist on the update source elements. If the type of the update source element is not to be traced for the given project, then no relation should exist on it. Due to mistakes of the developer or the tool, relations may exist anyway. Furthermore, there are rules defined within the catalog that copy or move impacted relations between different types of elements and so could also create unintended relations. To avoid traceability relations that would violate the current traceability information model, the traceability relation repository should validate any created or changed traceability relation against the project's traceability information model before performing the change. The customization of allowed traceability relations for a project should only be done within the traceability information model so as to define a project's traceability in one place.

5.3. Critique of Phase 2

The approach is intended to support the developer in the maintenance of traceability relations and to hence save tedious work. Even where interaction is required on the update, the user only has to make a few simple decisions instead of navigating through related models and searching for impacted traceability relations and their related elements manually.

Even where automated, it is important to inform the user about any automated changes to traceability relations. While the recognition of development activities happens in the background and goes unnoticed, the traceability update may be perceived. Such an indication helps the developer to understand the approach and allows for the validation of the development activity recognition and performed automated updates. The knowledge about missing updates allows traceability to be maintained manually where needed, to prevent its decay. By recognizing incorrect or missing updates, rule evolution is also possible.

6. Evaluation

This section describes an experiment that was undertaken to evaluate the approach with respect to the manual effort that can be saved and quality that can be reached with the maintained traceability while implementing changes to a specific development project. In the context of industrial use, there are two questions that matter when considering an automated solution for traceability maintenance. An adopter wants to know how much effort and cost can be saved by the proposed solution and whether the result, the maintained traces, is at least of comparable quality to that resulting from manual maintenance. Eventually, even an automated maintenance solution with significant effort savings will only be adopted, if the overall costs of traceability within a development project are less than the expected benefits. This break-even point is individual for each project and this overall analysis is not within the scope of this evaluation.

6.1. traceMAINTAINER prototype system

A prototype system called *traceMAINTAINER* has been implemented to help evaluate the approach. *traceMAINTAINER* works in the background while a developer works on software development using structural UML diagrams. It generates change events for elementary model changes and recognizes development activities. Once a development activity has been recognized, *traceMAINTAINER* performs the necessary maintenance to impacted traceability relations, with user interaction if necessary. Such user interaction is basically the only situation where an everyday user sees the prototype system. Fig. 8 shows that dialog for the development activity of moving a method between two classes. It requires the user to decide between alternative ways of updating the traceability relations. For setting-up and customizing the recognition and update process, the prototype system provides additional components that allow, for example, the editing and validation of rules. *traceMAINTAINER* is described more fully in Mäder et al. (2009b), Mäder et al. (2008c).

6.2. Industrial validation

We established a cooperation with Siemens in the Czech Republic in order to get information about the usefulness of the approach in industrial practice. The cooperating group fulfilled all the preconditions for applying the approach in their projects (i.e., model-based development using a UML modeling tool and having established traceability relations). The leader of this group agreed to apply the approach on two different projects over the course of one year.

This cooperation delivered valuable input for improving the approach and insight into traceability practice within the particular projects. Apart from technical feedback regarding improvement of the approach, we also received qualitative feedback on the

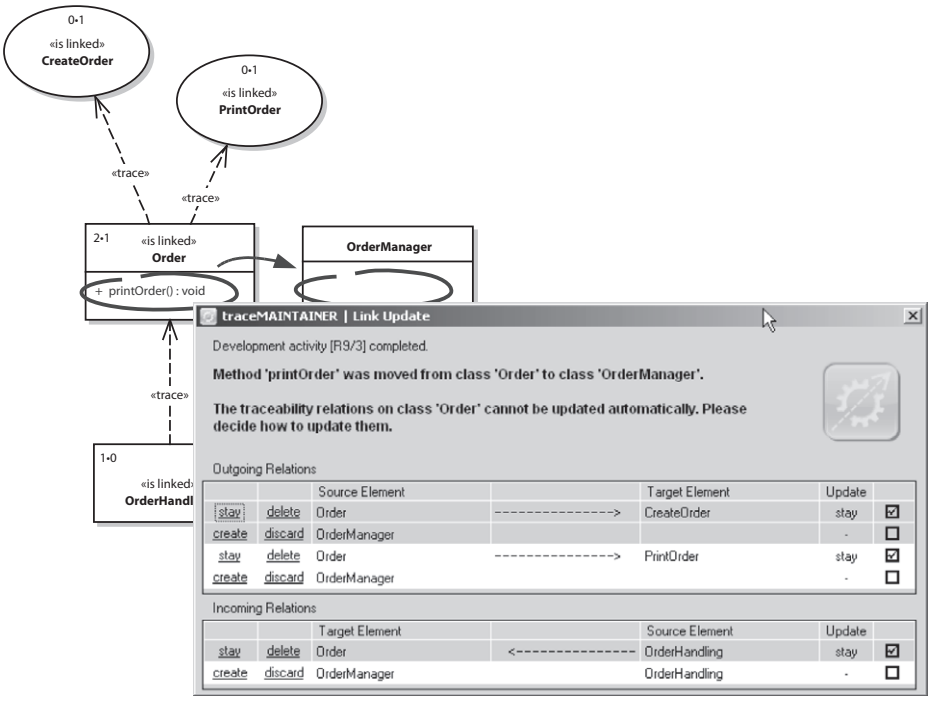


Fig. 8. Semi-automated traceability update with user interaction after moving a method between two classes

approach from practitioners at Siemens who were piloting the approach. Practitioners told us that they felt supported by the approach in maintaining traceability. Furthermore, they liked the fact that the approach was working automatically in the background, in most cases without user interaction. However, it delivered no measurable data. The reason was that the model under development was not accessible to people outside Siemens, so no baseline was available against which to compare results.

The controlled experiment described in the next section was developed in order to get measurable results.

6.3. Experimental set-up

We designed a controlled experiment in order to validate our approach. The experiment has one independent variable (the use of *traceMAINTAINER*) and two treatments (*tM*, *no − tM*). The aim of the experiment was to answer the two research questions described in the following.

6.3.1. Research question 1: manual effort

Does use of the approach reduce the manual effort necessary for maintaining traceability relations? While no manual effort would be a desirable target, evidence is sought of a reduction greater than the time necessary to configure and learn how to interact with *traceMAINTAINER* to make use of the approach worthwhile.

Measures: The manual effort for traceability maintenance refers to the time the developer spends on this task. It comprises: thinking about the maintenance task (including recognizing it); navigating within the models; and performing the required changes. Measuring the time taken for these sub-tasks is problematic given the lack of access to thought processes and the inter-weaved nature of the sub-tasks with the modeling tasks that cause the traceability maintenance to be required. Attempting to gain data about the complete traceability maintenance effort would require the user to provide additional information that allows distinguishing traceability maintenance from modeling. The intention of the selected tasks was to capture realistic data in a full modeling scenario without drawing the attention of our participants exclusively to traceability maintenance. As an indicator of effort, the number of performed changes to the set of traceability relations was recorded (manual changes for both treatments and automated changes for the *tM* treatment), along with the time spent responding to *traceMAINTAINER* dialogs. The dependent variables recorded in order to answer research question 1 were:

n_{M+}	Number of manually added relations to the link-set (the set of traceability relations).
n_{M-}	Number of manually deleted relations from the link-set.
n_{A+}	Number of automated added relations to the link-set.
n_{A-}	Number of automated deleted relations from the link-set.
n_{UI}	Number of user interactions with *traceMAINTAINER* before maintaining the link-set.
t_{UI}	Elapsed time between opening an interaction dialog and the user's response to it.

For subjects of the *no − tM* treatment, only n_{M+} and n_{M-} are available and were recorded, while for subjects of the *tM* treatment all variables were recorded.

Metrics: The number of manual changes to relations is the source for estimating the time and costs the developer spends on maintaining traceability. To be able to compare the number of manual changes among subjects of both treatments, the following metric compares the manual changes (cumulated added and deleted relations) for both treatments as a relative percentage of changes for the *tM* treatment:

$$rel.\, n_M(tM) = \frac{n_M(tM) - n_M(no - tM)}{n_M(no - tM)} \cdot 100 \quad (1)$$

where

$$n_M = n_{M+} + n_{M-} \quad (2)$$

The $rel.\, n_M(tM)$ metric does not take into account that manually adding a relation usually consumes more time then manually deleting a relation. Furthermore, the time that the user spends to react on a *traceMAINTAINER* update dialog t_{UI} is not taken into account. The time to undertake a manual change could not be measured precisely as discussed before.

For those reasons, we conducted a small experiment with ten subjects. Each subject was required to create ten new traces between given model elements, to remove ten existing traces, and

Table 1
Tested hypotheses to answer research question 1.

Variable	Null hypothesis H_0	Alternative hypothesis H_a
n_M	$n_M(tM) \geq n_M(no-tM)$	$n_M(tM) < n_M(no-tM)$

to respond to ten user interactions. Based on the results of that experiment, the comparable effort of the three direct measures was correlated as follows:

$$t_{M+} \approx 2 \cdot t_{M-} \approx 2 \cdot t_{UI}. \quad (3)$$

The correlation says that a user, on average, spends half the time on deleting a relation and on reacting to a *traceMAINTAINER* update dialog than she/he spends on creating a new relation. Using this relation to weigh the number of manual changes n_{M+} and n_{M-} performed by subjects of both treatments, and using the number of user interactions n_{UI} for the *tM* group, an approximate relative effort $rel.E(tM)$ for the *tM* treatment in relation to the $no-tM$ treatment can be computed as follows:

$$rel.E(tM) = \frac{n_{M+}(tM) + 0.5 \cdot n_{M-}(tM) + 0.5 \cdot n_{UI}(tM)}{n_{M+}(no-tM) + 0.5 \cdot n_{M-}(no-tM)} \quad (4)$$

Both metrics have been computed for the gathered data and are provided in Sections 6.4 and 6.5.

Hypothesis: The null hypothesis H_0 regarding research question 1 was that the number of manual changes to the set of traceability relations n_M performed by the experimental group *tM* is greater than or equal to the number of changes performed by the control group $no-tM$ (see Table 1). We expect that the use of the approach (independent variable) will reduce the number of manual changes to the set of traceability relations n_M (dependent variable) – this is the alternative hypothesis H_a. If the data gathered during the experiment supports the null hypothesis H_0 with a probability of 5% or less then this hypothesis would be rejected and the alternative hypothesis H_a considered.

6.3.2. Research question 2: maintenance quality

Do the traceability updates performed by the approach, in cooperation with the user where required, result in a set of traceability relations of comparable or better quality to those maintained manually?

Measures: An agreed baseline is required to determine the quality of a set of traceability relations. Here, the baseline refers to what would have been the correct and required changes to the traceability relations following the changes performed to the related models. For this experiment, the baseline was individual for each subject and her/his individual solution to the modeling tasks. These baselines were determined after the experiment by the experimenters in a labor intensive procedure. The guidelines for determining each baseline were the traceability information model and the task description given to the subjects. In cases of ambiguity, two experimenters decided about the correctness or incorrectness of a traceability relation. The experiment and baseline were prepared by the first author of the paper and by a graduate student, both with multi year experience in requirements engineering, model-based development, and requirements traceability. According to the baseline, three types of changes to the traceability relations were distinguished and represent the dependent variables captured to answer research question 2:

- Δ_c Changes that have been performed correctly according to the baseline.
- Δ_i Changes that have been performed incorrectly.
- Δ_m Missing changes that have not been performed.

Metrics: To be able to compare the correctness and completeness of the changes among the subjects and treatments, two metrics were computed that are commonly used to evaluate approaches dealing with uncertainty in recognition processes: precision and recall. Precision relates correct changes to all performed changes and informs the correctness of performed changes:

$$Q_P = \frac{\Delta_c}{\Delta_c + \Delta_i} \quad (5)$$

Recall relates correct changes to all required changes and informs the completeness of performed changes:

$$Q_R = \frac{\Delta_c}{\Delta_c + \Delta_m} \quad (6)$$

Table 2
Tested hypotheses to answer research question 2.

Variable	Null hypothesis H_0	Alternative hypothesis H_a
Q_P	$Q_P(tM) \leq Q_P(no-tM)$	$Q_P(tM) > Q_P(no-tM)$
Q_R	$Q_R(tM) \leq Q_R(no-tM)$	$Q_R(tM) > Q_R(no-tM)$

Hypothesis: The null hypothesis H_0 regarding research question 2 was that the values of the metrics Precision Q_P and Recall Q_R for the *tM* treatment are less than or equal to those of the $no-tM$ treatment (see Table 2). The alternative hypothesis H_a states that the precision and recall of changes are greater for the *tM* treatment.

6.3.3. Development project

The experiment was conducted on models for a mail-order system described as UML diagrams. These models described a completed project implemented in Java. The project had initially been developed as seminar work (Dommasch and Duhme, 2004) and was reverse engineered and enhanced into a complete model-based development project for the Enterprise Architect modeling tool. The project artifacts included models on three levels of abstraction: requirements, design and implementation. All diagrams of interest are shown in the appendices of Mäder (2009). The models provided information to a level of detail that one would expect at the end of the design phase, including use case diagrams, interaction diagrams and class diagrams. The diagrams and their elements are listed in Table 3. Please note that only the class and package diagrams were expected to be maintained by the subjects. Behavioral diagrams were not related by traces and, accordingly, were only available to provide the best possible information about the system to the subject. The set of traceability relations for this project relates the three models and consisted of 214 traceability relations. The initial linking was undertaken according to a traceability information model for the project that was also provided to the subjects of the experiment. This stated that only relations between requirements/design and design/implementation are valid (i.e., no intra model relations and no requirements/implementation relations). The relations are always directed from the dependent to independent model. Use cases and classes have to be related by at least one relation. Attributes, methods, components and packages can

Table 3
Project models, diagrams and elements.

	Requirements model	Design model	Implementation model
Diagrams			
Activity	7	–	–
Class	1	6	6
Package	–	1	1
State charts	3	–	–
Sequence	–	5	–
Use case	3	–	–
Elements			
Attributes	23	73	150
Classes	15	41	63
Methods	–	124	280
Packages	–	5	5

be related to any other element, as long as the preceding rules are followed.

6.3.4. Modeling tasks

Three maintenance tasks were to be performed on these models, in a fixed order, adding new features of practical value that would impact every part of the system. Although the underlying source code was made available within the implementation model, the tasks only required changes to the design and implementation models. Based on a pilot study with four additional master students, covering a spread of development experience, it was estimated that it would take 2–3 h to complete all the tasks. Subjects were permitted to perform the tasks according to their ideas and experiences to capture a realistic spread of different solutions to the same problem. This means that the solutions would not be comparable per se. The correctness analysis therefore required inspecting each individual model in order to find out about correct, incorrect and missing changes to the set of traceability relations. The subjects had to perform the following tasks:

1. Enhance the system's functionality to distinguish private and business customers and also to handle foreign suppliers.
2. Extract the two layers, *view* and *data*, of the system's three-layer architecture into separate components.
3. Enhance the system's functionality to handle additional product groups (e.g., *print media* and *consumer electronics*) and provide functionality to categorize products according to content categories.

The task descriptions and the questionnaires are available in the appendices of Mäder (2009).

6.3.5. Subjects

The subjects comprised 16 computer science students with a wide range of experience in UML and model-based software engineering. All the students were taking a course on software quality and were either in the 4^{th} or 5^{th} year of their diploma (Masters comparable). The subjects were partitioned into two groups of 8, based upon prior experience, to distribute expertise equally across the two treatments.

6.3.6. Data gathering

Data were gathered via three questionnaires. For both groups, a log file was created by *traceMAINTAINER* containing all the elementary changes performed by the subject, all changes to the link-set and information about how often the subject navigated the models using traceability relations. For the *tM* group, a log of all recognized development activities, the user decisions on interactions and all automatically performed traceability updates was also created. The models of all the subjects were available for the analysis.

6.4. Results

Fig. 9(a) depicts the number of manual changes n_M to the traceability relations for each single task and across all tasks for both treatments as a box plot. The information is also displayed, along with additional statistical measures, in Table 4, as variable $n_{M+} + n_{M-}$. The table shows, for each variable, the minimal (min), median, arithmetic mean and maximal (max) value separated according task and treatment. Furthermore, the standard deviation (sd) of the values within one treatment and the percentage difference between the mean values (% diff) of both treatments are depicted where appropriate. Fig. 9(b) depicts the number of responses to the *traceMAINTAINER* update dialog for each task and across all tasks for the *tM* treatment. This information is also displayed in Table 4 as variable n_{UI}. Fig. 9(c) and (d) depicts the measured precision Q_P and recall Q_R values for each task and across all tasks for both treatments as box plots. This information and additional statistical measures are also displayed in Table 4.

Univariate analyses of the dependent variables were performed to test the hypotheses, both individually for each task and across all tasks. For all the dependent variables n_M, n_A, n_{UI}, Q_P and Q_R, two-sample *t*-tests were performed. The preconditions of the *t*-test, normality and equality of variances, were given for the measured data. For normality, we tested with the one-sample Kolmogorov-Smirnov procedure and for the equality of variances we tested with the Levene's test. The level of significance for the hypotheses tests was set to $\alpha = 0.05$. P-values are provided in the *t*-test column of Table 4. One subject from each group had to be excluded from the analysis because they did not provide a minimal solution to each modeling task. This precondition was required in order to compare results between all subjects.

6.5. Discussion

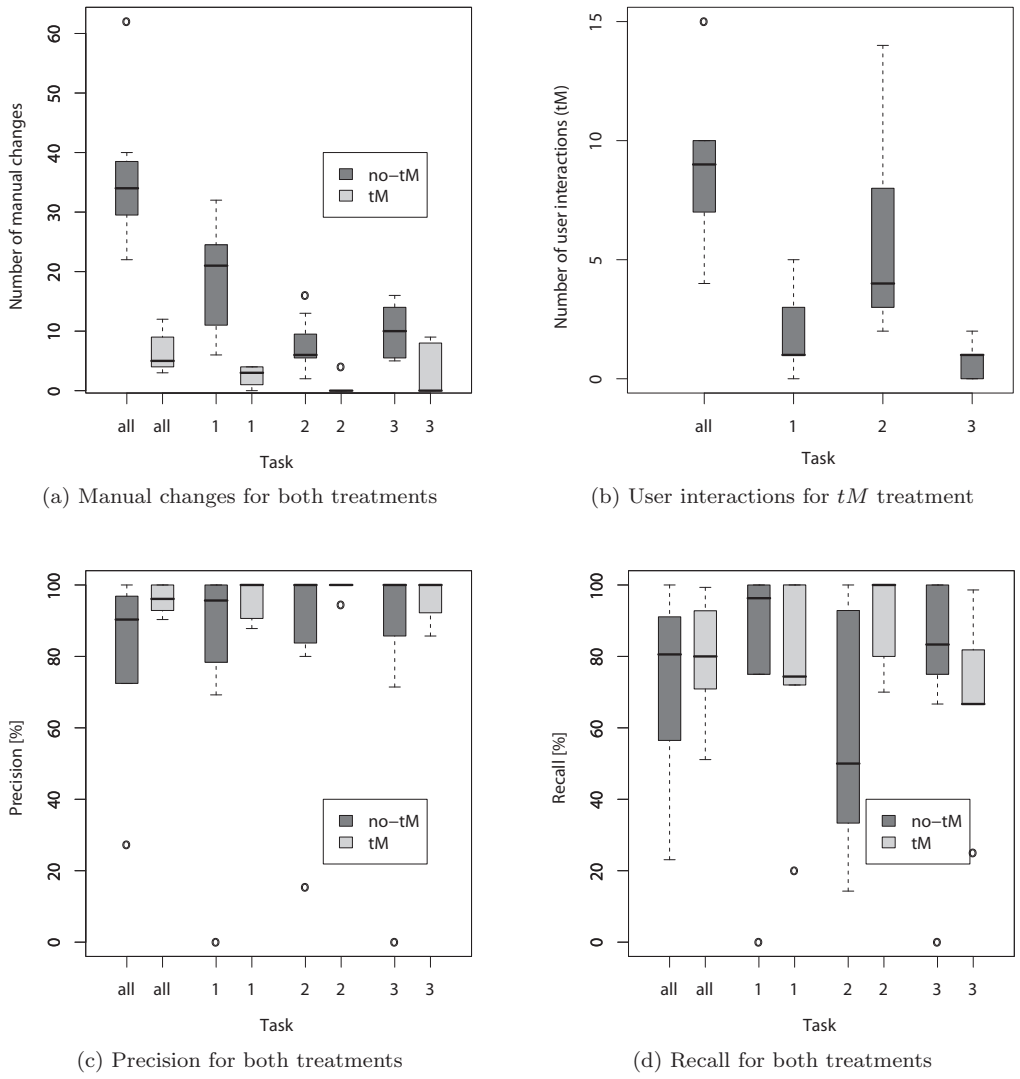
Within this section, the results presented in Section 6.4 are discussed separately in relation to the research questions.

6.5.1. Research question 1: manual effort

When examining the number of manual changes n_M to the link-set over the three tasks, the *tM* group performed far fewer changes ($rel.n_M(tM) = -82\,\%$) than the $no-tM$ group (see Table 4). This difference is statistically significant ($p-value < 0.01$). The result is similar for each single task.

Comparing the overall number of changes for both treatments (both manual and automated for the *tM* treatment), the figures show that the $no-tM$ group performed only half as many changes ($n_M(no-tM) = 36.3$) than the tM group's combined total of manual and automated changes ($n_M(tM) + n_A(tM) = 6.6 + 59.6 = 66.2$) for all tasks. The explanation for that behavior is that *traceMAINTAINER* recognizes small incremental change activities and updates traceability relations immediately (in the background) after recognition. This means that the changes reflect each detour of the developer, in contrast to manual maintenance where the update is typically performed after completing the whole task. Although the developer's strategy results in fewer changes and seems to be the better one, it is important to note that a developer is able to recognize semantic relations between model elements and to validate traceability relations at any time. In contrast, our automated approach discussed here has to follow each incremental change in order to update relations. The time to undertake a manual change could not be measured precisely because it is not clear when the developer starts to think about a change task. This effort was estimated indirectly via the manually created relations, the manually deleted relations and the user interactions. This measure provides an approximation to the saved effort by using *traceMAINTAINER*, as shown in Table 4. The values show that, across all tasks, the subjects of the *tM* treatment spent only $rel.E(tM) = 29\%$ as much manual effort on maintaining traceability as the subjects of the $no-tM$ treatment. The higher value of $rel.E(tM) = 52\%$ for task 2 results from the nature of the task that required a large number of move activities with impact on traceability relations of the parent elements (see Section 6.3.4). The traceability updates associated with these activities required a relatively large number of interactions with the user $n_{UI} = 6.2$ that, in combination with the low number of manual changes ($n_M(tM) = 0.8$ *vs.* $n_M(no-tM) = 7.7$), led to the higher relative effort of $rel.E(tM) = 52\%$ for task 2.

Overall, a decrease of 71% manual effort for the maintenance of traceability relations for the discussed experiment is seen as positive regarding the goal to reduce the manual effort for



(a) Manual changes for both treatments

(b) User interactions for tM treatment

(c) Precision for both treatments

(d) Recall for both treatments

Fig. 9. Results of the experiment as box plots. (a) Manual changes for both treatments (b) User interactions for *tM* treatment (c) Precision for both treatments (d) Recall for both treatments.

maintaining traceability relations, especially as the subjects of the *tM* treatment spent only a few minutes on their training session about the approach. To find out whether this decrease is large enough to facilitate a broader use of traceability and such (semi-) automated maintenance in practice remains to be evaluated in future industrial studies. After the experiment, all but one subject working with *traceMAINTAINER* stated that they felt comfortable with the technology after a short time and that they did not experience conflicts or disturbance by the automated updates. The remaining subject said that it was sometimes difficult to know whether an automated update would occur or whether a manual update was called for.

6.5.2. Research question 2: maintenance quality

The computed precision Q_P and recall Q_R provide information about the correctness and completeness of changes to the link-set (see Table 4). The results show that the *tM* group reached a precision $Q_P > 95\,\%$ for all tasks, with a low standard deviation ($sd(Q_P) = 2.5$–6.5%). This value is 21% higher than the average precision value of the $no - tM$ group across all tasks. Values of precision lower than 100% indicate that incorrect changes have been performed to the link-set. We analyzed these incorrect changes and found that among all the changes performed by *traceMAINTAINER*, no incorrect ones were found. All incorrect changes within the *tM* group were manual changes of the subject. Please note that our approach automates only the maintenance of existing traceability, but the general nature of our tasks required also the addition of completely new elements and accordingly the creation of new traces. Precision and recall measure the overall quality of all link changes performed during the experiment.

Recall of changes measures whether all necessary changes have been performed. Across both groups and all tasks, not all required link changes were performed (all recall values lower than 100%) showing that subjects of both treatments missed additionally required manual link changes. The average values per task are more diverse than the precision results. For tasks 1 and 3, the $no - tM$ group reached higher values of recall (6 and 11%) than the *tM* group. An analysis of the results showed that all our rules fired as intended and indicates that subjects working with our automated approach relied on that support and missed more of the additionally required manual changes. We wanted to explore the capabilities of the approach without putting the focus of subjects too much on our approach. Accordingly, we did not explain all concepts of the approach, nor how and when automated updates are to be expected. That approach might have actually influenced the

Table 4
Number of changes to traceability relations and number of user interactions during traceability update (upper table) and precision and recall of changes to traceability relations (lower table).

Task	Variable	Treatment	Min	Median	Mean	Max	sd	% diff	*t*-Test	Rel. effort
1	$n_{M+}+n_{M-}$	$no-tM$	6	21	18.6	32	9.5	−87%	<0.01	18%
		tM	0	3	2.4	4	1.8			
	$n_{A+}+n_{A-}$	tM	0	38	36.2	65	23.9			
	n_{UI}	tM	0	1	2.0	5	2.0			
2	$n_{M+}+n_{M-}$	$no-tM$	2	6	7.7	16	4.9	−90%	0.01	52%
		tM	0	0	0.8	4	1.8			
	$n_{A+}+n_{A-}$	tM	8	14	13.0	17	3.3			
	n_{UI}	tM	2	4	6.2	14	4.9			
3	$n_{M+}+n_{M-}$	$no-tM$	5	10	10.0	16	4.7	−66%	0.04	33%
		tM	0	0	3.4	9	4.7			
	$n_{A+}+n_{A-}$	tM	2	7	12.4	37	14.0			
	n_{UI}	tM	0	1	0.8	2	0.8			
All	$n_{M+}+n_{M-}$	$no-tM$	22	34	36.3	62	12.8	−82%	<0.01	29%
		tM	3	5	6.6	12	3.8			
	$n_{A+}+n_{A-}$	tM	19	51	59.6	114	34.7			
	n_{UI}	tM	4	9	9.0	15	4.1			

Task	Variable	Treatment	Min	Median	Mean	Max	sd	% diff	*t*-Test
1	Q_P	$no-tM$	0.0	95.7	78.9	100.0	36.5	21%	0.34
		tM	87.8	100.0	95.7	100.0	6.0		
	Q_R	$no-tM$	0.0	96.3	78.0	100.0	36.3	−6%	0.82
		tM	20.0	74.4	73.3	100.0	32.7		
2	Q_P	$no-tM$	15.4	100.0	83.3	100.0	31.0	19%	0.29
		tM	94.4	100.0	98.9	100.0	2.5		
	Q_R	$no-tM$	14.3	50.0	59.5	100.0	35.3	51%	0.10
		tM	70.0	100.0	90.0	100.0	14.1		
3	Q_P	$no-tM$	0.0	100.0	81.6	100.0	37.5	17%	0.44
		tM	85.7	100.0	95.6	100.0	6.5		
	Q_R	$no-tM$	0.0	83.3	76.2	100.0	35.8	−11%	0.67
		tM	25.0	66.7	67.8	98.6	27.3		
All	Q_P	$no-tM$	27.3	90.3	79.5	100.0	25.7	21%	0.19
		tM	90.3	96.1	95.9	100.0	4.3		
	Q_R	$no-tM$	23.1	80.6	71.3	100.0	27.6	11%	0.61
		tM	51.1	80.0	78.8	99.3	19.0		

quality results of the *tM* group, as they did not exactly know when to expect automated updates and when they were required to update manually. What we learned from that finding is that users require a good tutorial on when to expect automated maintenance and when they are required to create relations manually. For task 2, the $no-tM$ group had a recall of only 59.5% versus 90% in the *tM* group. That task required large structural changes to the model (see Section 6.3.4) with high impact on the traceability relations of elements and their parent elements. A possible explanation is that the subjects working manually were not able to remember all the pre-existing traceability relations before their changes in order to re-establish them after the changes. This explanation would correlate with results from the inspection of the models after the experiment.

However, any differences between the two treatments for both measures are not statistically significant (compare column *t*-test in Table 4). Nevertheless, the aim of the approach was to provide maintenance quality that is comparable to that reached by the developer manually.

6.5.3. Dependence on the subjects' experience

In addition to the previous analyses, an analysis of variances (ANOVA) exploring the combined effect of the treatment and the subjects' experience on the number of manual changes to the link-set n_M, the precision of changes Q_P and the recall of changes Q_R was performed. The aim was to find out whether the subjects' experience had also a significant impact on the results of the experiment. We performed separate ANOVAs for each dependent variable and for each task as well as across all tasks. Table 5 shows the p-values of all these analyses. The values within the confidence level of $\alpha = 0.05$ are marked in bold.

The results show that the number of manual changes depends, for each single task and across all tasks, upon the treatment (see p-values in the treatment row of Table 5). This finding has already been discussed in Section 6.5.1 and there is no significant difference in precision and recall depending on the treatment (see Section 6.5.2). Regarding the dependence of the three variables (n_M, Q_P and Q_R) upon the experience of the subject (see p-values in the experience row of Table 5), the impact was not significant, except for the recall of task 1. For task 1, the experience of the subject had a significant impact on the recall Q_R the subjects reached with their changes. This means that more experienced subjects reached a better recall Q_R of their changes to traceability relations than less experienced subjects here. This correlation has only been found for task 1 and a p-value of 0.05 shows only a weak significance. Overall, the analyses show that the experience of the subjects had almost no impact on the results (number of manual changes to the link-set n_M, precision of changes Q_P and recall of changes Q_R). To be able to generalize these findings, additional experiments with a larger population of subjects would be required.

6.6. Threats to validity

This section discusses what is considered to be the most important threats to the validity of the discussed experiment.

Table 5
Analysis of variance (ANOVA) of the treatment and the experience of the subject on the dependent variables.

Independent variable	Dependent variable											
	Manual changes n_M				Precision Q_P				Recall Q_R			
	All	1	2	3	All	1	2	3	All	1	2	3
Treatment	**0.00**	**0.00**	**0.02**	**0.05**	0.17	0.32	0.32	0.42	0.58	0.79	0.12	0.65
Experience	0.25	0.11	0.57	0.67	0.17	0.21	0.78	0.23	0.12	**0.05**	0.68	0.14

6.6.1. External validity

The issue of external validity concerns whether a causal relationship holds over variations in persons, settings, treatments and outcomes that were either in the experiment or not. From a task perspective, the reported experiment is realistic. Students (future young professionals) were working on a real project, using commercial tools and implementing demanding tasks. We used the Sparx Enterprise Architect modeling tool, which is commercially available and has a very large (>100k), globally distributed user base. A brief survey on the tracing functionality of UML tools showed that it is comparable among common modeling tools. Nevertheless, it is hard to draw conclusions to a wider population without more studies. The results reflect more a tendency that shows the potential of the approach. There are threats associated with the short time the subjects spent on the experiment (3 h) given the task complexity. However, it was not the focus of the experiment to set trivial tasks with obvious changes. A high-level task description enabled a variety of ways to solve the problem, but demanded effort to analyze and evaluate the data (55k lines of log messages and 16 different models). Without sophisticated techniques, it would be complicated to run an experiment lasting much longer.

6.6.2. Internal validity

Internal validity is concerned with establishing a causal relationship, here between the use of *traceMAINTAINER* and the number of manual changes to the link-set. Based upon prior experience, we created two groups with equal distribution regarding the experience factor. One group was assigned to the *tM* treatment, the other to the *no – tM* treatment, creating a balanced design. In order to have comparable results among participants, it was necessary to exclude one subjects from each treatment after the experiment since they did not solve the modeling tasks completely, making their results not comparable. The potential influence of the facilitators was addressed by providing an initial briefing and task description in written form only. The difference in the material between groups was marginal, the addition being how to react on a user interaction for the *tM* group. None of the subjects had any prior knowledge of the approach nor did they know the experimental goals.

6.6.3. Construct validity

Construct validity refers to having established correct operational measures for the constructs being studied. To investigate the effect of the approach on the effort for maintaining traceability relations after the evolution of related UML models it is necessary to use the UML as intended. The UML offers an open set of description techniques with many ways to apply them. In this experiment, six types of diagram at different levels of detail were used, the subjects had state-of-the-art education in UML development and a widely distributed CASE tool was used. The debriefing interview showed that almost all the subjects became familiar with the tool after their prior tutorial. The examination of the resulting models showed that, except for two cases (explained before), all the subjects were able to edit and evolve the UML models in a manner comparable to industrial practice.

To investigate the effort for maintaining traceability relations, we decided for an indirect measure, i.e., counting the number of manual link changes and user interactions. To emulate a realistic setting we demanded model maintenance tasks, not traceability maintenance tasks, so we were unable to distinguish the time spent on traceability maintenance from the remaining time for solving a task. The disadvantage of that decision is that we cannot compute concrete time differences based upon our data, only the relative differences in effort. We found that trade-off acceptable, though further studies could focus on concrete time differences.

To investigate the correctness of the changes to the link-set, the main problem with comparing traceability is the lack of an agreed standard. Therefore, a traceability information model was provided to give guidance on how to establish traceability for the project. The initial creation of traceability relations was done according to the model and the subjects were required to do likewise. In order to gain comparable results, further restrictions were made as to the minimal number and direction of relations (see Section 6.3).

6.6.4. Reliability

It is expected that replications of the experiment should offer results similar to those presented in this section. Of course, concrete measured values will differ from those presented here as they are specific to the subjects, but the underlying trends and implications should remain unchanged as the analysis showed that the experience of the subject had little influence on the results. Furthermore, pilot studies executed with two additional developers on two projects (described in Mäder et al., 2008a,b), showed similar results and so support the findings.

7. Critical review and future work

While seeking evaluation partners for the approach, two interesting perspectives arose when talking with practitioners from Siemens who were responsible for traceability within development projects. After an explanation of the approach, one individual replied: “I do not think that I like an automated solution for this sensible task. I want people to think about their changes again while maintaining traceability.” A second individual replied: “that is exactly the solution I was waiting for, it can save us a lot of work”. This reflects the fact that the decision for or against a (semi-) automated approach to traceability maintenance is individual. The developed approach can save tedious and error-prone work, but it should not be seen as a solution that makes the maintenance of traceability relations something that the developer does not have to think about anymore, as there may be intrinsic value in that activity. Potential limitations of the approach are examined in this concluding section, with obvious implications for future research.

7.1. Assumptions of the approach

Section 3.2 lists the assumptions that were made while developing the approach. These assumptions refer to the development

process of a project and require model-based development, using a UML modeling tool and the application of traceability in accordance to a traceability information model. Except for the traceability information model, this is exactly the scenario that has been reported by nine of the ten interviewed companies in a survey paper (Mäder et al., 2009c). The extension of the approach to additional types of diagrams and models remains a future task. Regarding use of a traceability information model, a major problem is the missing support for such definitions within common CASE tools (Ritze, 2008). For the evaluation of the approach, the support of a traceability information model has been integrated into the prototype tool.

7.2. Predefined rule catalog

A limitation of the approach is that only predefined development activities can be recognized and these are unlikely to reflect all possible development approaches, so it will be necessary to customize and extend the rule catalog. In order to address this issue, a rule editor is provided with sophisticated checks to validate changes to rules. Nevertheless, this task remains a manual one.

7.3. Scope of and threats to empirical studies

It is a future exercise to gain more statistical data on the cost/benefit trade-off of the approach, costs in terms of customizing and extending the rules, and benefits in terms of the time saved on manual maintenance across all projects using the rules. The discussed experiment showed a substantive saving of manual effort while using the existing rule catalog. Unfortunately, no information is available about how much work of a development project accurately relates to evolving related models and triggering the necessity for traceability maintenance. Such information can only be gained by performing studies in real projects. There is clearly a need for empirical work here and gaining that information should be a future research topic for the traceability community. While we cannot provide concrete figures, we can refer to the often-used figures that claim that 70% of a development's budget is spent on software maintenance (Glass, 2002). Where this is a model-based development process, we assume a considerable amount of effort that is spent on changing and updating models. Without concrete information, it is problematic to understand how much effort is really saved for a project, despite best approximations.

7.4. Semantic correctness of relations

The approach maintains existing traceability relations whether they are semantically correct or not. It is not possible to find out about the correctness of relations or even to improve their quality via the approach. This means that it requires a reasonable set of initial traceability relations to make the approach useful. For projects where this initial quality cannot be guaranteed, the manual maintenance of traceability relations might be the better choice, allowing the developer to correct problems when recognized.

7.5. Uncertainty in the recognition process

There are several points of uncertainty in the process for recognizing development activities that might lead to incorrect or missing traceability updates. Missing rules and missing alternatives within rules can lead to unrecognized development activities and missing updates, while insufficiently defined rules can lead to the recognition of development activities that have not been performed and so to incorrect traceability updates. The validation functionality within the rule editor supports the identification of certain problems within a rule definition, but a proportion of ensuring the correctness and completeness of the rule catalog remains manual work.

7.6. Future work

One goal during the development of the approach was to reduce the manual effort involved in the maintenance of traceability relations as much as possible. There are two areas that still require manual work: selecting impacted traceability relations during a semi-automated update; and customizing and extending the rules. Regarding the selection of impacted traceability relations during a semi-automated update, the visualization and animation of the impacted element, along with all related elements before and after the update could support the user in her/his decision. Regarding the definition of rules, the existing rule editor could be extended by functionality that allows new rules to be defined semi-automatically by observing a developer performing change activities in situ while using a rule recorder. Furthermore, a more intuitive, preferably visual, representation of rules would be desirable in order to facilitate an easier rule definition and customization by users. Finally, the extension and improvement of the rule catalog is a continuous effort that requires ongoing attention as more results from industrial usage become available. For example, whether there should be a default update for each deletion or addition to a composition or inheritance structure remains an open issue and we plan to work on it in the future.

While the support of structural UML diagrams addresses a common scenario in industry (Mäder et al., 2009c), it would be desirable to extend the approach to other kinds of development model. The necessary preconditions would be models described in a semi-formal language with a defined meta-model and sufficient element properties to allow for the identification of meaningful development activities. For example, the following types of diagram would meet these preconditions to a certain extent: behavioral UML diagrams, feature diagrams, Mathworks Simulink™ diagrams and NI LabView™ diagrams.

The information that is gained about performed development activities by the user is currently only used to enable the maintenance of traceability, but this information could also support other activities within the development process. It could, for example, be used to generate change logs for version control systems and could also support the checking of correctness of the performed development activities.

The propagation of changes was another goal of the approach and has been incorporated into the recognition and update process. Nevertheless, it was not the main focus of this work and a more sophisticated and selective procedure based on the available information about the change and its type, following the work of Cleland-Huang et al. (2003), could be possible.

The requirements traceability problem (Gotel and Finkelstein, 1994) has many facets and it is unlikely that there will ever be a single approach that solves the whole problem, but much work has been done on this topic over the past decade that provides promising approaches to partial aspects. A major goal for the traceability community should clearly be the integration of these techniques in order to provide a solution for the whole traceability life cycle of a project. Two examples illustrate the advantages an integration of existing approaches with the one discussed in this paper could offer. First, the integration with approaches that support the initial creation of traceability relations would allow existing projects to retain their investment in traceability by ongoing (semi-) automated maintenance of the generated relations. Second,

by integrating keyword matching techniques it would be possible to provide more support for the developer in creating new traceability relations as a project evolves. A dialog could provide possible counterparts for new traceability relations of an element, according to the traceability information model, ranked using keyword matching techniques.

Another more technical issue of integration relates to the tooling scenario within larger projects. Different artifacts of the development process are often held in different tools and the support for traceability between these tools varies (Mäder et al., 2009c). This integration seems to be a precondition for the extended use of traceability in larger industrial projects and thus for its maintenance.

Acknowledgements

The authors thank Tobias Kuschke and Christian Kittler for implementing the prototype, Johannes Langguth for his work in preparing and analyzing the experiment, and all the students who were involved in the experiment. Furthermore, the authors would also like to thank the anonymous reviewers of an earlier version of this paper for their useful suggestions for improving the final paper. This research was part funded by DFG grant Ph49/7-1 and by the Austrian Science Fund (FWF): M1268-N23.

References

Aizenbud-Reshef, N., Nolan, B.T.,Rubin, J., Shaham-Gafni, Y., 2006. Model traceability. IBM Systems Journal 45 (3), 515–526, ISSN 0018-8670, http://dx.doi.org/10.1147/sj.453.0515.

Alexander, I., 2002. Toward automatic traceability in industrial practice. In: Proceedings of 1st International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE02), Edinburgh, UK, pp. 26-31, http://www.soi.city.ac.uk/zisman/WSTPapers/Alexander.pdf.

Antoniol, G., Canfora, G., Casazza, G., Lucia, A.D., Merlo, E., 2002. Recovering traceability links between code and documentation. IEEE Transactions on Software Engineering 28 (10), 970–983, ISSN 0098-5589, http://www.computer.org:80/tse/ts2002/e0970abs.htm.

Arkley, P., Riddle, S., 2005. Overcoming the traceability benefit problem. In: Proceedings 13th International Requirements Engineering Conference, IEEE Computer Society, pp. 385–389, ISBN 0-7695-2425-7.

Arlow. J., Neustadt, I., 2005. UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design, second edn. Addison-Wesley, ISBN 0-321-32127-8.

Cleland-Huang, J., Chang, C.K., Ge, Y., 2002. Supporting event based traceability through high-level recognition of change events. In: Annual International Computer Software and Applications Conference (COMPSAC02), IEEE Computer Society, pp. 595–602, ISBN 0-769-51727-7, http://doi.ieeecomputersociety.org/10.1109/CMPSAC.2002.1045069.

Cleland-Huang, J., Chang, C.K., Christensen, M.J., 2003. Event-based traceability for managing evolutionary change. IEEE Transactions on Software Engineering 29 (9), 796–810, ISSN 0098-5589, http://csdl.computer.org/comp/trans/ts/2003/09/e0796abs.htm.

Coleman, D., 1994. Object-Oriented Development: The Fusion Method. Prentice-Hall, ISBN 0-131-01040-9.

Dömges, R., Pohl, K., 1998. Adapting tracability environments to project-specific needs. Communications of the ACM 41 (12), 54–62, 0001-0782.

Dommasch, C., Duhme, D., 2004. Versandhandel: Hausarbeit in Objektorientierte Analyse und Design. Tech. Rep., http://www.christian-dommasch.de/downloads/versand/versand-dokumentation.pdf.

Du, D., Gu, J., Pardalos, P.M. (Eds.), 1997. Satisfiability Problem: Theory and Applications, vol. 35 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, ISBN 0-821-80479-0.

Egyed, A., Grünbacher, P., Heindl, M., Biffl, S., 2007. Value-based requirements traceability: lessons learned. In: Proceedings 15th International Requirements Engineering Conference (RE07), pp. 115–118, ISSN 1090-705X.

Engels, G., Heckel, R., Küster, J.M., Groenewegen, L., 2002. Consistency-preserving model evolution through transformations. In: Proceedings 5th International Conference UML 2002 – The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools, vol. 2460 of Lecture Notes in Computer Science. Springer, Dresden, Germany, ISBN 3-540-44254-5, pp. 212-226.

Finkelstein, A.C.W., Gabbay, D.M., Hunter, A., Kramer, J., Nuseibeh, B., 1994. Inconsistency handling in multiperspective specifications. IEEE Transactions on Software Engineering 20 (8), 569–578, ISSN 0098-5589.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley, ISBN 0-201-48567-2.

Glass, R.L., 2002. Facts and Fallacies of Software Engineering. Addison-Wesley, Boston, MA.

Gotel, O.C.Z., Finkelstein, A.C.W., 1994. An analysis of the requirements traceability problem. In: Proceedings of the First International Conference on Requirements Engineering (ICRE94), IEEE Computer Society, Colorado Springs, CO, pp. 94-101, ISBN 0-8186-5480-5.

Hayes, J.H., Dekhtyar, A., Osborne, J., 2003. Improving requirements tracing via information retrieval. In: Procerdings of 11th IEEE International Requierments Engineering Conference (RE03), IEEE Computer Society, pp. 138-148, ISBN 0-7695-1980-6, doi:10.1109/ICRE.2003.1232745.

Hnatkowska, B., Huzar, Z., Kuzniarz, L., Tuzinkiewicz, L., 2003. Refinement relationship between collaborations. In: Proceedings Workshop on Consistency Problems in UML-based Software Development, UML'03, IEEE Computer Society, San Francisco, USA, pp. 51–57.

Huzar, Z., Kuzniarz, L., Reggio, G., Sourrouille, J.-L., 2004. Consistency problems in UML-based software development. In: Nunes, N.J., Selic, B., da Silva, A.R., Álvarez, J.A.T. (Eds.), UML Satellite Activities, vol. 3297 of Lecture Notes in Computer Science. Springer, pp. 1–12, ISBN 3-540-25081-6.

Jacobson, I., Rumbaugh, J., Booch, G., 1999. The Unified Software Development Process, Object Technology Series. Addison-Wesley, Reading, MA, ISBN 0-201-57169-2.

Kruchten, P., 2000. Rational Unified Process: An Introduction. Addison-Wesley, Reading, MA, ISBN 0-201-70710-1.

Lano, K., 2005. Advanced Systems Design With Java, UML and MDA. Elsevier Butterworth-Heinemann, Amsterdam, The Netherlands, ISBN 0-750-66496-7.

Lucia, A.D., Oliveto, R., Tortora, G., 2008. IR-based traceability recovery processes: an empirical comparison of one-shot and incremental processes. In: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15–19 September 2008, L'Aquila, Italy, IEEE Computer Society, pp. 39-48, ISBN 978-1-4244-2776-5, doi:10.1109/ASE.2008.14.

Luckham, D., 2002. The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley Professional, Reading, MA, ISBN 0-201-72789-7.

Mäder, P., Gotel, O., Philippow, I., 2008a. Rule-based maintenance of post-requirements traceability relations. In: Proceedings of 16th International Requirements Engineering Conference (RE'08), Barcelona, Spain, pp. 23–32, ISSN 1090-705X.

Mäder, P., Gotel, O., Philippow, I., 2008b. Enabling automated traceability maintenance by recognizing development activities applied to models. In: Proceedings of 23rd International Conference on Automated Software Engineering (ASE2008), L'Aquila, Italy.

Mäder, P., Gotel, O., Kuschke, T.I., 2008c. Philippow traceMaintainer – automated traceability maintenance. In: Proceedings of 16th International Requirements Engineering Conference (RE'08), Barcelona, Spain, pp. 329–330.

Mäder, P., Gotel, O., Philippow, I., 2009a. Enabling automated traceability maintenance through the upkeep of traceability relations. In: Proceedings 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA2009), Enschede, Netherlands.

Mäder, P., Gotel, O., Philippow, I., 2009b. Semi-automated traceability maintenance: an architectural overview of traceMAINTAINER. In: Proceedings 5th ECMDA Traceability Workshop (ECMDA-TW 2009). In conjunction with the 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA2009), Enschede, Netherlands.

Mäder, P., Gotel, O., Philippow, I., 2009c. Motivation matters in the traceability trenches. In: Proceedings of 17th International Requirements Engineering Conference (RE'09), Atlanta, Georgia, USA.

Mäder, P., Gotel, O., Philippow, I., 2009d. Getting back to basics: promoting the use of a traceability information model in practice. In: Proc. 5th Int'l Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE2009), Vancouver, Canada.

Mäder, P., 2009. Rule-based maintenance of post-requirements traceability. Ph.D. thesis, Technische Universität Ilmenau.

Maletic, J.I., Collard, M.L., Simoes, B., 2005. An XML based approach to support the evolution of model-to-model traceability links. In: Proceedings of 3rd International Workshop on Traceability in Emerging Forms of Software Engineering TEFSE'05, ACM, New York, NY, USA, pp. 67–72, ISBN 1-59593-243-7, http://doi.acm.org/10.1145/1107656.1107671.

Marcus, A., Maletic, J.I., 2003. Recovering documentation-to-source-code traceability links using latent semantic indexing. In: Proceedings of the 25th International Conference on Software Engineering (ICSE03), IEEE Computer Society, Piscataway, NJ, pp. 125-137, http://computer.org/proceedings/icse/1877/18770125abs.htm.

Mens, T., van der Straeten, R., Simmonds, J., 2005. A framework for managing consistency of evolving UML models. In: Yang, H. (Ed.), Software Evolution with UML and XML. IGI Publishing, Hershey, PA, USA, pp. 1–30, ISBN 1-591-40462-2.

Murta, L.G.P., van der Hoek, A., Werner, C.M.L., 2006. ArchTrace: policy-based support for managing evolving architecture-to-implementation traceability links. In: 21st IEEE/ACM International Conference on Automated Software Engineering, ASE '06, pp. 135–144, doi:10.1109/ASE.2006.16.

Murta, L.G.P., van der Hoek, A., Werner, C.M.L., 2008. Continuous and automated evolution of architecture-to-implementation traceability links. Automated Software Engineering Journal 15 (1), 75–107, ISSN 0928-8910, http://dx.doi.org/10.1007/s10515-007-0020-6.

OMG, 2003. MDA Guide Version 1.0.1. Object Management Group (OMG), Framingham, MA, http://www.omg.org/mda,omg/2003-06-01.

OMG, 2008. OMG System Modeling Language (OMG SysML) Version 1.1. Object Management Group OMG, Framingham, MA, http://www.omg.org/spec/SysML/1.1,formal/2008-11-01.

OMG, 2010. OMG Unified Modeling Language Specification (OMG UML) Version 2.3. Object Management Group (OMG), Framingham, MA, http://www.omg.org/spec/UML/2.3/.
Pinheiro, F.A.C., 2004. Requirements traceability. In: Leite, J.C.S.P., Doorn, J. (Eds.), Perspectives on Software Requirements. Kluwer Academic Publishers, The Netherlands, pp. 91–113, ISBN 1-402-07625-8.
Ritze, M., 2008. Comparison of the Traceability Functionality of CASE-Tools. Tech. Rep., Technical University of Ilmenau, Ilmenau, Germany.
Royce, W.W., 1987. Managing the development of large software systems: concepts and techniques. Reprinted in: Proceedings of the 9th international conference on Software Engineering ICSE '87. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 328–338, ISBN 0-89791-216-0.
Russek, L., 2004. OpenQuasar development of open source components with the aid of quasar. In: Dadam, P., Reichert, M. (Eds.), INFORMATIK 2004 – GI Jahrestagung, vol. 2. GI, pp. 488–492, ISBN 3-885-79380-6.
Shen, W., Lu, Y., Low, W.L., 2003. Extending the UML metamodel to support software refinement. In: Proceedings of the Workshop on Consistency Problems in UML-Based Software Development. In conjunction with UML2002, IEEE Computer Society, San Francisco, USA, pp. 35–42.
Siedersleben, J., 2004. Moderne Software-Architektur. Dpunkt-Verlag, ISBN 3-898-64292-5.
Spanoudakis, G., Zisman, A., Pérez-Mi nana, E., Krause, P., 2004. Rule-based generation of requirements traceability relations. Journal of Systems and Software 72 (2), 105–127, ISSN 0164-1212, http://dx.doi.org/10.1016/S0164-1212(03)00242-5.
van Bemmel, J., Dockhorn, P., Widya, I., 2004. Paradigm: Event-driven Computing, White paper TI/RS/2004/051. Lucent Technologies, CTIT, https://doc.telin.nl/dscgi/ds.py/Get/File-48190.
Weilkiens, T., 2006. Systems Engineering mit SysML/UML. Dpunkt-Verlag, ISBN 3-898-64409-X.

Dr. Patrick Mäder received a Diploma degree in industrial engineering and a Ph.D. degree (Distinction) in computer science from the Ilmenau University of Technology in 2003 and 2009, respectively. He worked as a consultant for the EXTESSY AG, Wolfsburg between 2003 and 2005. Currently, he is a postdoctoral fellow at the Institute for Systems Engineering and Automation (SEA) of the Johannes Kepler University, Linz. Dr. Mäder also has an active collaboration with the Software and Requirements Engineering Center at DePaul University, Chicago. His research interests include topics related to software engineering, with a focus on requirements traceability, and object-oriented analysis and design.

Dr. Orlena (Olly) Gotel has been active in the area of traceability for over 20 years and received a Ph.D. on the topic from Imperial College, University of London, in 1995. Dr. Gotel has published widely on different aspects of the traceability problem and is currently an Officer of the Center of Excellence for Software Traceability, where she is coordinating the Grand Challenges of Traceability and working on its Body of Knowledge. In addition to academic research and teaching positions in the UK (Oxford University, City University, University College London) and the US (Pace University), Dr. Gotel has held senior positions within the UK defense industry working on Systems Requirements Engineering. Dr. Gotel received a B.Sc. (Hons) in Computer Science from the University of Warwick in 1989 and a M.Sc. (Distinction) in Advanced Methods in Computer Science from the University of London in 1990.