traceMAINTAINER: A Tool for the Semi-automated Maintenance of Model Traceability

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

[1] Department of Software Systems
Ilmenau Technical University, Germany
patrick.maeder|ilka.philippow@tu-ilmenau.de
[2] Department of Computer Science
Pace University, New York, USA
ogotel@pace.edu

Abstract. *traceMAINTAINER* is a tool that supports an approach for maintaining post-requirements traceability relations after changes have been made to traced model elements. The update of traceability relations is based upon predefined rules, where each rule is intended to recognize a development activity applied to a model element and trigger the necessary traceability update directives. This means that little manual effort or interaction with the developer is required to keep traceability relations up to date as a development process proceeds. *traceMAINTAINER* can be used with a number of commercial software development tools and enables the semi-automated maintenance of traceability relations stored within these tools. This paper provides technical details of *traceMAINTAINER*'s architecture and major components.

Keywords: Development activity recognition, evolutionary change, model traceability, rule-based traceability maintenance, traceability.

1 Introduction

Traceability relations support stakeholders in understanding the dependencies between artifacts created during the development of a software system. Traceability relations thus enable engineers to perform many development-related tasks, such as: (a) confirming the implementation of requirements; (b) analyzing the impact of changing requirements; and (c) supporting regression tests after changes. To ensure that all these benefits can be realized, it is necessary to have a complete and correct set of traceability relations between the established artifacts. That goal requires, not only the creation of the relations during the initial development process, but also the maintenance of these relations after changes to the associated artifacts. The large number of relations that potentially exist even for a small development project demands effective method and tool support for these tasks. Several authors (Gotel and Finkelstein [1], Ramesh and Jarke

[2], Arkley et al. [3]) found the necessity to establish and maintain traceability manually to be the main reason for its rare use in industrial projects.

traceMAINTAINER is a prototype tool that supports the semi-automated update of traceability relations between requirements, analysis and design models of software systems expressed in UML ([4], [5]). This is made possible by analyzing elementary 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 development activities are expressed as predefined rules. Rules consist of masks requiring an event with certain properties. Once recognized, the corresponding rule gives a directive to update impacted traceability relations to restore consistency. *traceMAINTAINER* can enable developers to update traceability relations with little manual effort. This approach to traceability maintenance has been described fully in [4] and [5]. A high-level overview of the architecture of *traceMAINTAINER* has also been provided in [6]. This current paper specifically provides more extensive details on the components and implementation of *traceMAINTAINER*.

The *traceMAINTAINER* software has been implemented with Microsoft Visual Studio .Net and supports the following general scenarios: the analysis of a flow of change events according to a set of predefined rules imported from a XML rule catalog; the execution of necessary traceability updates after development activities have been recognized; and the editing and validation of existing rules, along with the specification of new rules.

To support these scenarios, *traceMAINTAINER* consists of multiple components (see Figure 1) and is designed with the objective of creating an implementation that is as independent as possible from specific third-party UML modeling tools. The central component, the rule engine, handles the development activity recognition and computes the maintenance directives. It provides an interface for receiving new change events and requires an interface for querying and updating traceability relations. The rule engine is deployable with any tool that allows the necessary change events to model elements to be captured, and that allows its traceability relations to be established and changed from outside the tool.

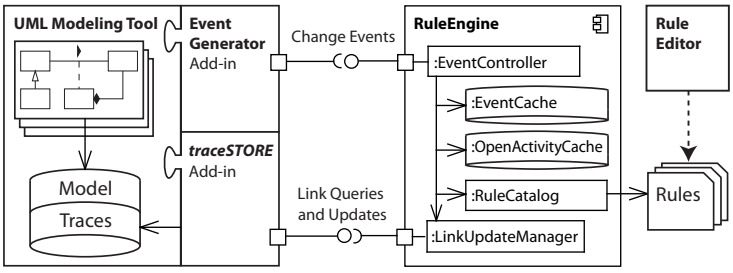

Fig. 1. Overview of the *traceMAINTAINER* components

The change event interface is designed to be used by a tool-specific event generator that recognizes changes to model elements and collects data about the changed model element in order to create change events for the rule engine. The required query and update interface also has to be implemented by a tool-specific adapter. Depending on where the traceability relations are stored, the adapter gives access to relations stored within the tool's model or to an external relationship repository. For the prototype, the additional *traceSTORE* repository has been implemented. *traceSTORE* enhances the third-party UML modeling tool used and stores relations within its model, but provides for significantly more functionality in terms of traceability than the modeling tool itself. In addition, a rule engine reads a rule catalog stored in XML format. This catalog can be edited and validated with a specific rule editor. Each of the major components of *traceMAINTAINER* is described in detail in the following sections of the paper.

2 Rule Engine

The rule engine is the main component of *traceMAINTAINER*. It consists of an *EventCache* that holds a number of last incoming change events from the event generator, an *OpenActivityCache* that holds all partly recognized activities for the events currently held in the *EventCache* and a *LinkUpdateManager* that determines the necessary traceability maintenance actions for recognized development activities. It depends upon the rule catalog for its operation (see Section 5).

The rule engine has a single user interface intended for the normal user. This is the interaction dialog that automatically pops-up in situations where a development activity has been recognized, but not enough information is available to carry out the traceability update completely automatically (see Figure 2). The dialog provides detailed information about the recognized development activity and the necessary update. The dialog text can be customized within the rule catalog as all the properties of events assigned to the recognized activity can be used to build this description (e.g., names and types of changed model elements).

The dialog shows two list boxes separating the incoming and outgoing traceability relations involved in the update context. Each row in these boxes represents an existing or potential new traceability relation. Pictograms depict existing relations and reflect decisions on desired update actions. They distinguish relations that will remain, those that will be created and those that will be deleted during the update depending on the decision of the user. Relations with the same element on the target side are grouped together using the same background color. A group shows the relations residing on one or more update source elements and allows the user to create these on the update target element(s). The user can decide to keep (stay is the default and preselected action) or delete existing relations on the source elements. For the target elements, the user can decide to create or discard the relation. A decision on the proposed relations without preselected actions determined is required to be able to complete the update (i.e., all the boxes on the right hand side of the relations must be checked). In

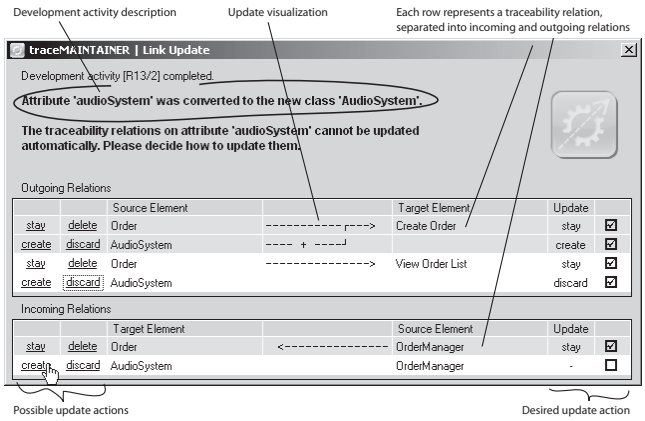

Fig. 2. The user interaction dialog is displayed in situations where a traceability update cannot be carried out automatically

addition, a change of a proposed action by *traceMAINTAINER* is made possible, but not required. Tooltips are provided when the user hovers over the elements to provide additional information about elements and relations.

For more advanced users interested in the underlying approach, the rule engine provides a window that allows the user to observe the current status of the *EventCache* and *OpenActivityCache* (see Figure 3). The window consists of three list boxes. The left box gives a view of the events currently in the *EventCache*. The right box shows all the partly recognized activities in the *OpenActivityCache*. The middle box shows properties of events or masks depending on the selected item in the *EventCache* (left box) or *OpenActivityCache* (right box).

The view on the *OpenActivityCache* (right box) shows all the masks of currently open activities and their current state. A mask might have already been assigned to an event currently in the *EventCache*, so a click on this mask highlights the assigned event in the *EventCache* (left box) and shows a merged list of expected event properties defined within the mask and the actual assigned properties of the event in the middle box. A mask might not yet be assigned, but be comparable to incoming events. A click on such a mask shows the required properties within the mask for a matching event (middle box). A mask might not yet be comparable, because references to other masks cannot be resolved as those other masks are not yet assigned to an event. Additional functions within the status window allow the user to reset both caches and to add events stored in a file to the cache for testing purposes.

A menu further gives access to the settings, grouped into three categories regarding the *EventCache*, the recognition process and the update functionality. The settings associated with the *EventCache* allow the user to set the size of the *EventCache* and to customize the functionality to persistently store all events

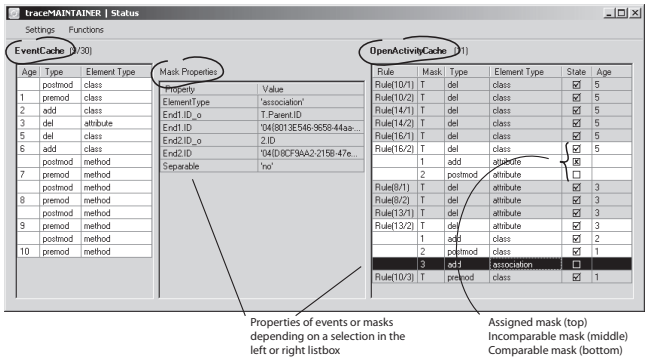

Fig. 3. The status window allows the *EventCache* and *OpenActivityCache* of the rule engine to be observed

within the cache upon closing the current model and to restore them when re-opening. The settings associated with the development activity recognition allow the user to require a notification on each recognized activity, even if a fully automated update of relations is possible. This supports demonstration and debugging tasks. The settings associated with traceability update allow the user to switch off the update when testing the recognition part of the prototype and to configure the user interaction dialog for situations where either all the relations already exist on the update target element or only one relation already exists on the update source element, thereby reducing user interaction to a minimum. The rule engine also has extensive logging functionality that allows the user to analyze and re-execute captured modeling scenarios.

3 Event Generator

As highlighted in the introduction, not all parts of *traceMAINTAINER* are tool-independent. The event generator is tool-dependent and the version that is discussed in this paper has been created as an add-in to Sparx Enterprise Architect [7]. After an evaluation of current third-party UML modeling tools, Enterprise Architect was chosen due to its usability, extensibility, ease of installation and low price. Its traceability support is comparable to that of other such tools, so stereotyped dependency relations are intended to be used as traceability relations in accordance with the UML meta-model.

The event generator add-in observes changes to elements of interest and captures a number of properties to the changed element. The types of element to be observed and their properties of interest are defined within an accompanying information model stored in XMI format. The information model can be opened as a regular model in Enterprise Architect and in other UML tools to allow the user to customize the generated events in terms of observed elements and collected properties.

The implementation of how to observe changes to certain model elements and how to find property values is specific to the modeling tool, but all the remaining aspects like reading the information model, communicating with the rule engine and the architecture of the add-in remains common among different adapters. A reference implementation is provided to support the use of other tools.

4 Relationship Repository *traceSTORE*

The access to the traceability relationship repository is also a tool-specific part of *traceMAINTAINER*. If traceability relations are created and stored by using the functionality of the third-party UML modeling tool itself (e.g., Enterprise Architect) then an additional adapter is necessary to make these relations accessible for the rule engine of *traceMAINTAINER* so as to query and change relations. Where traceability relations are stored in an external repository, separated from the actual model, then an adapter between the rule engine and that repository is required. Such an adapter has been implemented for the EXTESSY ToolNET traceability repository [8].

Both solutions have their advantages. If traceability relations are stored directly with the model itself then it is easier to maintain consistency, whereas the external repository allows relations to be stored that connect elements of different tools (in the case of ToolNET). However, the traceability functionality of modeling tools may not allow for all types of model elements to be related (e.g., attributes, methods and associations are not traceable in Enterprise Architect) and the handling of traceability relations may be rudimentary. ToolNET provides more functionality but obviously requires the user to use an additional tool.

An additional relationship repository was created to evaluate the approach supported by *traceMAINTAINER*. The *traceSTORE* is like the event generator implemented as an add-in for Enterprise Architect. It provides an additional menu within Enterprise Architect that allows the user to store traceability relations connected to any kind of model element within an extra class model that extends the related model. This approach prevents inconsistencies that are likely to occur between a model and a separately stored set of traceability relations. The *traceSTORE* context menu of a model element provides features to see and navigate to the associated elements of an existing relation, to start or end a new relation, and to delete an existing relation as required. The subjects of our experiments and our industrial evaluation partners have reported that this tight integration with the modeling tool assists traceability even when performed manually ([4], [5]).

The creation of a new traceability relation incorporates three steps. After selecting a source and a target element, a dialog is shown visualizing the desired relation. This allows the user to choose a different relation type to the default *'trace'* type. After committing, *traceSTORE* creates two classes representing references to the two related elements and a dependency relationship between them in the extra class model. Each reference contains the identifier, name, type

and model of the original element. A stereotype storing the type of the relation is then attached to the dependency relationship.

traceSTORE addresses another requirement that arose while developing the approach for semi-automated traceability maintenance. All related elements are enhanced with indicators showing the current number of incoming and outgoing traceability relations on an element. This feature enables immediate feedback to the user after automated changes have been performed to traceability relations by *traceMAINTAINER* for ongoing status visibility.

In addition to creating, deleting and navigating through traceability relations, *traceSTORE* allows the user to import traceability relations that have already been created and stored within a tool's model into *traceSTORE*, to export relations from *traceSTORE* back into the model, and to check the consistency between the model and the relations in *traceSTORE*. For related elements, it validates they still exist within the model and updates additional stored information. For relations, it validates and updates their depicted count of related elements within the model, and further checks for false relations like self-links, multiple relations of the same type between the same elements, and false or missing types of relations. Note that the consistency check is only necessary after a malfunction of *traceSTORE*, if the model is being evolved without *traceSTORE* being enabled, or if the user performs manual changes to the *traceSTORE* class model.

Upon opening a new model within the modeling tool, *traceSTORE* reads the project-sepcific traceability information model that provides information about permitted and required traceability relations. Within *traceSTORE*, this information model is used to validate any intended new traceability relation regardless of whether created manually by the developer or as part of a semi-automatic traceability update. This means that only defined traceability relations can be created manually by the user or as part of the automated traceability updates of *traceMAINTAINER*.

5 Rule Editor and Rule Catalog

The rule editor (see Figure 4) is a stand-alone application that is intended to help in two use scenarios.

First, it validates an existing rule catalog upon opening it according to four categories of possible failures: the structure of the rule catalog's XML file is validated against an XSD schema definition; the element types and properties defined within masks are validated against the information model that defines events to be generated and the information contained within them (see Section 3); the syntax of required values for properties is validated against regular expressions also defined within the information model; and all references are validated for the existence of the referenced element within the definition of the current rule.

Second, it provides functionality to edit and create all parts of a rule catalog whilst also validating the changes. For rules, the update and the description

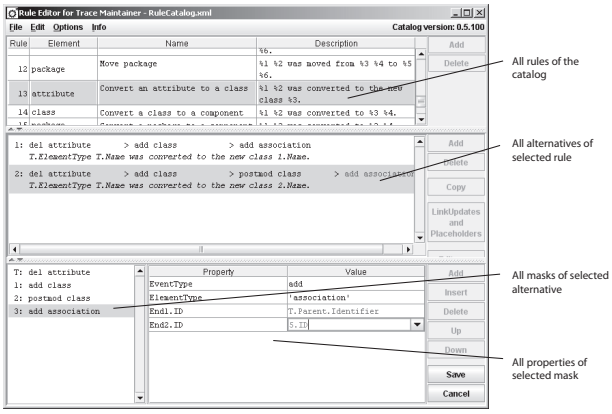

Fig. 4. The rule editor allows the rule catalog to be customized while validating the entries

provided within notifications can be customized (see Section 2). Each alternative can have an informal description supporting the comprehension of the catalog for later changes. An alternative consists of exactly one *TriggerMask* and a number of additional masks. Each mask has a type of change (i.e., add, delete, pre-modification or post-modification) and a number of required properties, all part of the information model. The required values of properties can be defined as static values or references to properties of other masks of the alternative. Boolean expressions are supported to allow logical combinations of several values. To support definition, typical values for each property can be stored in the information model and provided within a drop-down box in the rule editor. Each entry made for a property is syntax-checked and all defined references are checked for their existence within the alternative. Failures are indicated by highlighting the syntax mistakes and reference mistakes.

6 Scenario of Use

A simple scenario is provided to illustrate the use of *traceMAINTAINER*: a change to a requirement impacts a realized use case and it becomes necessary for the developer to convert an existing attribute within one class into its own class (see Figure 5).

Step 1 of the figure shows the initial situation and the traceability relation between class *Order* and use case *Create Order*. Steps 2 to 5 show one way for the developer to carry out the development activity based on a sequence of elementary changes. With the last elementary change, deleting the original attribute, the development activity is recognized by *traceMAINTAINER* and the necessary update of traceability relations is performed automatically. Step 6 shows the automatically created traceability relation between class *AudioSystem* and

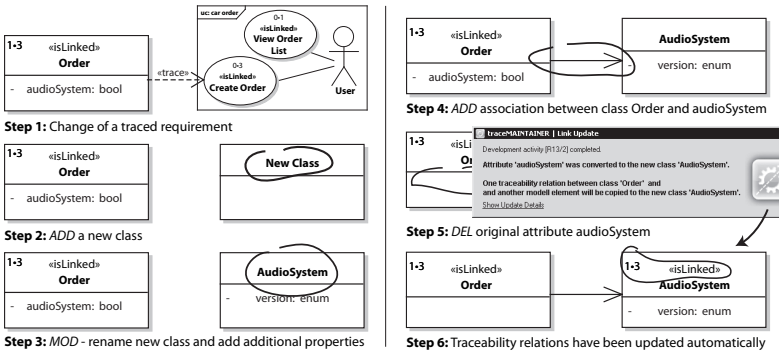

Fig. 5. Development activity with semi-automated traceability maintenance

use case *Create Order*. *traceMAINTAINER* can recognize the same development activity via any ordering of the same elementary changes or via different sets of change events.

The scenario is representative of most development activities. Nevertheless, there are activities that, although recognized by *traceMAINTAINER*, do not lead to clear directives for traceability update. In such situations, the dialog depicted in Figure 2 is shown after recognizing a development activity and so it requires the user to decide upon the update for certain traceability relations.

7 Status

traceMAINTAINER provides an extensive set of features for implementing and maintaining traceability between a broad spectrum of UML model element types. The underlying approach is intended as a complement to those approaches that initially create a set of traceability relations using manual or automated techniques. Limitations of the approach are that only predefined activities can be recognized and these are unlikely to reflect all possible development approaches, so it is necessary to customize the rules to project specifics.

The prototype tool's main component, the rule engine, has been implemented independent of a specific UML modeling tool and supplies a well-defined API. In this paper, we have described tool-specific extensions for the Enterprise Architect modeling tool to satisfy the API and further enhance its existing traceability functionality. We have created a rule editor for the definition, customization and validation of rules to allow for easy evolution. The recognition part of the approach required a complex and sophisticated implementation. To ensure the quality of the implementation while the prototype was evolving, an extensive set of automated unit tests was created and executed during the build process. In addition, captured scenarios of elementary change events with known results provide black-box and integration tests that were executed before releasing new versions to testers and users.

The current version of *traceMAINTAINER* is a second completely reengineered and restructured revision of all components. This version has proven stable in two industrial evaluation projects at Siemens. Initial experimental results have been encouraging ([4], [5]) and further industrial case studies are planned. Recently, a larger experiment with sixteen subjects, eight of them working with *traceMAINTAINER* and eight of them performing manual traceability maintenance was carried out. All the subjects had to perform the same modeling tasks on a given development project over a period of three hours. Analysis of the resulting models and the captured changes, performed either manually or supported by *traceMAINTAINER*, showed that the subjects of the *traceMAINTAINER* treatment spent approximately 71% less manual effort on maintaining traceability relations than the subjects within the group that performed unsupported manual maintenance. The quality of the maintenance was comparable among both treatments [9].

We are currently investigating how to semi-automatically define rules by observing a developer performing change activities in situ using a rule recorder. We are further investigating how to handle the undo function within third-party UML modeling tools effectively, whilst still recognizing development activities accurately.

Acknowledgments The authors would like to thank Tobias Kuschke, Christian Kittler and Arne Roßmanith for implementing the prototype.

References

1. Gotel, O.C.Z., Finkelstein, A.C.W.: An analysis of the requirements traceability problem. In: First International Conference on Requirements Engineering (ICRE'94), IEEE CS Press (1994) 94–101
2. Ramesh, B., Jarke, M.: Toward reference models of requirements traceability. IEEE Trans. Software Eng **27**(1) (2001) 58–93
3. Arkley, P., Mason, P., Riddle, S.: Position paper: enabling traceability. In: Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering, Edinburgh, UK (September 2001) 61–65
4. Mäder, P., Gotel, O., Philippow, I.: Rule-based maintenance of post-requirements traceability relations. In: Proceedings of 16th International Requirements Engineering Conference (RE'08), Barcelona, Spain (September 2008) 23–32
5. Mäder, P., Gotel, O., Philippow, I.: Enabling automated traceability maintenance by recognizing development activities applied to models. In: Proc. of 23rd Int'l Conf. on Automated Software Engineering ASE2008, L'Aquila, Italy (September 2008)
6. Mäder, P., Gotel, O., Philippow, I.: Semi-automated traceability maintenance: An architectural overview of tracemaintainer. research demo. In: Proc. 31st Int'l Conf. on Software Engineering ICSE2009, Vancouver, Canada (May 2009)
7. Sparx Systems: Enterprise Architect. `www.sparxsystems.com.au`
8. Extessy AG: Extessy ToolNet – Traceability Tool. `www.extessy.com`
9. Mäder, P., Gotel, O., Philippow, I.: Enabling automated traceability maintenance through the upkeep of traceability relations. In: Proc. Fifth European Conf. on Model-Driven Architecture FA ECMDA2009, Twente, The Netherlands (June 2009)