traceMaintainer – Automated Traceability Maintenance

Patrick Mäder[1], Orlena Gotel[2], Tobias Kuschke[1] 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 maintains post-requirements traceability amongst the elements of structural UML models. The maintenance of traceability relations is based upon predefined rules. Each rule recognizes a development activity applied to a model element.* traceMaintainer *carries out associated traceability updates in the background after an activity has been completed, requiring minimal manual effort and limited interaction with the developer. Currently,* traceMaintainer *can be used with two commercial software development (CASE) tools to update the traceability relations stored within them, while the underlying approach extends further to maintaining traceability within a heterogeneous and distributed environment of tools.*

1 Introduction and Motivation

Up-to-date traceability relations support many software development tasks, such as validating the implementation of requirements and analyzing the impact of changing requirements. This support requires not only the creation of traceability relations during initial development, but also the maintenance of these relations after any changes are made to related artifacts. The large number of potential relations, even for small systems, demands effective method and tool support.

We focus on post-requirements traceability between requirements and subsequent development artifacts [1] and have developed an approach to automatically maintain such relations established between elements of structural UML models [2]. The traceMaintainer tool supports this approach. It analyzes elementary change events captured while working within a CASE tool. Within the captured flow, sequences of events are searched that correspond to predefined rules that specify recurring development activities, semantically meaningful changes to a model composed of elementary changes. A matching rule results in a directive to update impacted relations to restore traceability.

2 *traceMaintainer*

traceMaintainer has been designed to be deployable with any CASE tool that allows for the capture of the necessary elementary change events and permits for the manipulation of traceability relations from outside the tool. It is necessary to write an adapter for each tool. The main purpose of an adapter is to generate change events and collect element properties to provide the rule engine with standardized data for processing. Adapters also allow the rule engine to update the traceability relations kept within the tool. We have developed adapters to ARTiSAN Studio and Sparx Enterprise Architect to date, and have created a rule catalog for changes to structural UML models developed within these tools (Figure 1). The rules are defined using XML according to a XML Schema Definition.

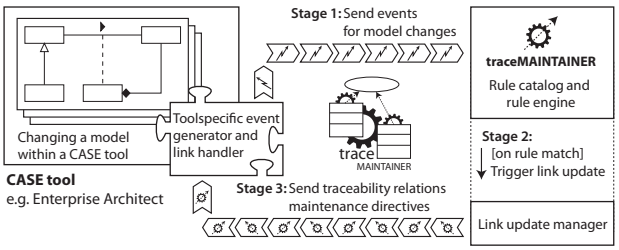
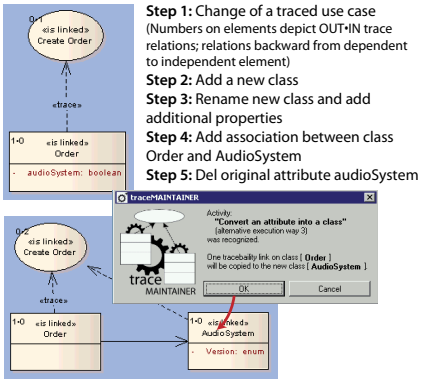


Figure 1. traceMaintainer's architecture

In heterogeneous settings of requirements and software engineering tools, the CASE tools are only used to capture the necessary change events. The traceability relations of the composite toolset are maintained within a third-party tool. EXTESSY ToolNet has been used as the repository for our work in this area. traceMaintainer handles the rule matching and provides ToolNET with the update directives.

3 Scenario

We provide a simple scenario to illustrate our approach and tool support. A change to a requirement impacts a realized use case and it becomes necessary for the developer to convert an attribute into its own class (Figure 2). Step 1 shows the initial situation and the 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 change event, 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 use case *Create Order*. traceMaintainer can recognize the same development activity via any ordering of the same elementary changes or via different sets. A matching algorithm makes it unnecessary to specify variations.



Step 1: Change of a traced use case (Numbers on elements depict OUT•IN trace relations; relations backward from dependent to independent element)
Step 2: Add a new class
Step 3: Rename new class and add additional properties
Step 4: Add association between class Order and AudioSystem
Step 5: Del original attribute audioSystem
Step 6: Traceability links have been updated automatically (2 incoming links on use case, 1 outgoing link on each class)

Figure 2. Updating traceability relations

The scenario is representative of most change activities. Nevertheless, there are some activities that, although recognized by traceMaintainer, do not lead to clear directives for traceability update. Figure 3 shows an example where method *printOrder* has been moved from class *Order* to class *OrderManager*. Class *Order* holds two traceability relations to realized use cases and one traceability relation to associated source code (see upper left corner of class *Order*). For this activity, it is not possible to determine whether the moved method is part of the realization of one or both of the realized use cases. In such ambiguous situations, traceMaintainer shows the dialog depicted in Figure 3 to let the user decide between traceability update alternatives.

traceMaintainer distinguishes traceability to independent model elements (outgoing relations) and to dependent model elements (incoming relations). Here, class *Order* has one incoming relation from source code and two outgoing relations to use cases. Both incoming and outgoing relations can be updated by traceMaintainer automatically, but it is likely that the dependent model element is no longer valid given its dependency upon the now changed element. For this reason, incoming traceability relations are set as *suspect* after automatic update to facilitate manual inspection.

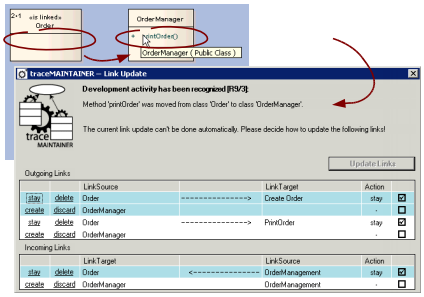


Figure 3. Decisions on traceability update

4 Status

The success of our approach depends upon tool support and the quality of the predefined rules. To define a comprehensive set of rules, we studied several development methodologies and industrial projects, and determined the traceability-relevant activities that typically occur during the analysis and design of systems using UML. 38 development activities have been identified and are part of traceMaintainer's rule catalog (21 rules with 67 alternatives). These have been used, validated and refined, and the early results are encouraging [2].

Since the rules are likely to evolve, we are creating an editor for their definition and validation. We are also investigating how to semi-automatically define rules by observing the developer performing change activities in situ using a rule recorder. We are further investigating how to handle the undo function within CASE tools effectively, whilst still recognizing development activities accurately, and industrial case studies are planned.

References

[1] O. C. Z. Gotel and A. C. W. Finkelstein. An analysis of the requirements traceability problem. In *1st Int'l Conf. Req. Eng. (ICRE)*, pages 94–101. IEEE CS, 1994.

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