Rule-Based Maintenance of Post-Requirements Traceability Relations

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

An accurate set of traceability relations between software development artifacts is desirable to support evolutionary development. However, even where an initial set of traceability relations has been established, their maintenance during subsequent development activities is time consuming and error prone, which results in traceability decay. This paper focuses solely on the problem of maintaining a set of traceability relations in the face of evolutionary change, irrespective of whether generated manually or via automated techniques, and it limits its scope to UML-driven development activities post-requirements specification. The paper proposes an approach for the automated update of existing traceability relations after changes have been made to UML analysis and design models. The update is based upon predefined rules that recognize elementary change events as constituent steps of broader development activities. A prototype traceMaintainer has been developed to demonstrate the approach. Currently, traceMaintainer can be used with two commercial software development tools to maintain their traceability relations. The prototype has been used in two experiments. The results are discussed and our ongoing work is summarized.

Keywords: Change; Post-requirements traceability; Rule-based traceability; Traceability maintenance.

1 Introduction and problem statement

Traceability relations articulate the dependencies between artifacts created during a software systems development project. These relations help stakeholders to undertake many development tasks, such as: (a) verifying the implementation of requirements; (b) analyzing the impact of changing requirements; (c) retrieving rationale and design decisions; and (d) supporting regression testing after changes have been made. To ensure that such benefits accrue, it is necessary to have an accurate (i.e., representative and up-to-date) set of traceability relations between the artifacts, established at a level of granularity that is suitable for the project at hand. This requires not only the creation of the relations during the initial development process, but also the maintenance of these relations after changes have been made to the associated artifacts. The number of potential traceability relations, even for small software systems, demands effective method and tool support.

The need to manually establish and maintain traceability relations, and the difficulty to achieve these tasks automatically as a by-product of development activities, is a recognized reason for the limited use and effectiveness of traceability in industrial projects (Ramesh and Jarke [16], Arkley et al. [3]). Much of the recent academic research has focused on the first aspect of this problem through the automated generation of traceability relations. The majority of these approaches apply text mining and information retrieval techniques to identify candidate relations and have been producing promising results (Alexander [1], Antoniol et al. [2], Marcus and Maletic [14], Huffman Hayes et al. [12]). However, they often require manual effort to ensure correct relations have been identified. One way to address this issue is for competing techniques to vote and reach consensus [8]. Research addressing the maintenance aspect of the problem has been less extensive. Spanoudakis et al. [19] describe rules based on information retrieval for the automated creation and subsequent maintenance of traceability relations. Cleland-Huang et al. [5] describe an approach for maintenance which informs the owner of artifacts about relevant changes to requirements so they can take action. Related work is discussed in Section 6.

The approach described in this paper focuses exclusively on the maintenance of traceability relations during the evolution and refinement of structural UML models. The initial creation of the traceability rela-

tions between these artifacts could be established manually or automatically. Whichever way, the approach is concerned with sustaining the investment that has been made in creating the initial set. The approach revolves around the monitoring of elementary changes that take place to UML model elements within a CASE tool and the generation of change events based upon these, using a set of rules to help recognize these events as constituent parts of intentional development activities. This is in recognition that the process of refining an analysis model towards a final design model, and any evolution of these models resulting from changing requirements, comprises a number of recurring development activities. Once these activities have been identified, traceability relations related to the changing model elements can be updated automatically.

The remainder of the paper is structured as follows. In Section 2, we present an overview of our approach. In Section 3, we discuss the rules used to maintain traceability relations. In Section 4, we describe the prototype developed to implement our approach. In Section 5, we present the results of two experiments to provide preliminary validation. In Section 6, we give an account of related work and discuss the contribution of our work. Finally in Section 7, we highlight outstanding issues that are the subject of future work.

2 Approach

For traceability to realize its promise of development task support, it is necessary to have an accurate set of traceability relations for a project. Due to the relatively low precision of the candidate relations generated by automated techniques, and the manual intervention often required to prune these, it is not viable to regenerate traceability relations automatically every time either a change is made or they need to be used. The ultimate vision would be the ability to establish and maintain traceability relations concurrently with development activities in an automated manner so they are always ready to use.

Our work supports part of this vision by focusing on maintaining traceability as a by-product of changes made to structural UML models during object-oriented software development. An assumption of our approach is that development activities that are part of this process are undertaken within a Computer Aided Software Engineering (CASE) tool. We focus on structural UML models and in maintaining post-requirements traceability [11] as this a common scenario in industry. Our approach also assumes the pre-existence of an initial set of traceability relations established between the models. Given these preconditions, we support the following scenarios: (i) the change of a model within the same level of abstraction (e.g., evolving the analysis model), typically to align the model as a result of changing or new requirements; and (ii) the change of a model into a more detailed level of abstraction (e.g., refining the analysis model into the design model), typically to explore requirements realization.

Our approach consists of three stages:

1. capturing changes to model elements and generating elementary change events (Section 2.1);
2. recognizing the wider development activity applied to the model element, as comprised several elementary change events (Section 2.2); and
3. updating the traceability relations associated with the changed model element to maintain accuracy of the set (Section 2.3).

Figure 1 illustrates these stages using an example that replaces an unspecified association in a UML design model with two unidirectional associations. This activity is undertaken as most programming languages are not able to implement bidirectional associations [4], so is done for requirements realization (scenario (ii) above). The resulting two associations are semantically equivalent to the preceding single association, meaning that all original traceability relations should be reestablished at both resulting elements. Stage 1 on the right hand side of Figure 1 shows a flow of elementary change events: deleting an unspecified association between class *Order* and class *Customer*, creating a unidirectional association between class *Order* and class *Customer* and creating a unidirectional association between class *Customer* and class *Order*. These changes allow for recognizing the refinement development activity shown in stage 2. This triggers the update of the existing and single traceability relation by copying it to the resulting two elements in stage 3.

2.1 Capturing elementary change events

Some CASE tools allow the capture of changes to UML models as change events. For our approach, we focus on the UML classifiers and relations that establish the structure of the system as model elements of interest: class, component, package, attribute, method, association, dependency, inheritance and stereotypes of these (e.g., aggregation, composition, association class and interface). We distinguish three types of change to these elements: add, delete and modify.

We also maintain information about the properties of those model elements that the changes are applied to (e.g., name and identifier). For the addition of an element, these properties only exist after the creation of

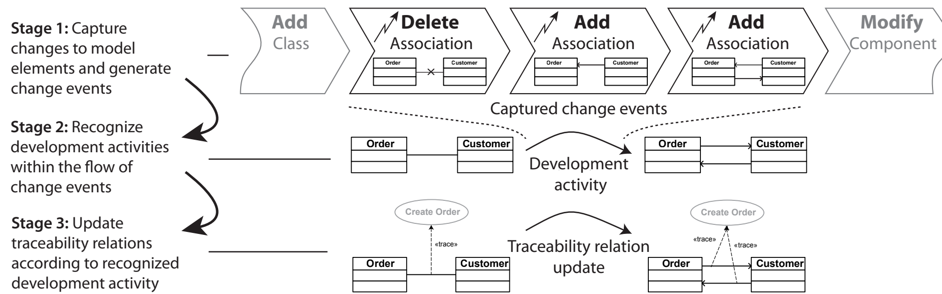

Figure 1. Stages of the approach as visualized by the example of refining an association

the element, and for deletion they only exist before destruction. For the modification of an element, both pre and post modification properties are required for analysis. We define four elementary change events: ADD, DEL, preMOD and postMOD. Stage 1 of our approach captures the change type applied to a model element and generates the corresponding change event(s) with all the necessary properties of the changed element.

2.2 Recognizing development activities

It is not possible to maintain traceability relations by only examining elementary change events. Stage 2 of our approach has to recognize the overarching development activity that is realized by a collection of elementary change events. We therefore search for predefined patterns within a flow of change events while working with a CASE tool. These correspond to the constituent steps of development activities that can be carried out while changing a UML model. The task of recognizing these involves challenges that are handled by the approach. (a) Several elementary change events can relate to one activity. The type of a change and the impacted model element do not offer enough information to relate changes to each other. It is necessary to compare additional properties. For the example of Figure 1, the replacing associations have to be directed, in opposite directions and between the same classes as the original. (b) The same development activity can be achieved by different elementary changes. For Figure 1, rather than deleting the unspecified association and adding two new unidirectional associations, it could be modified to a directed association. (c) The same development activity can be achieved by the same elementary changes in different sequences. For Figure 1, the replacing unidirectional associations could be established before the deletion of the existing association.

2.3 Maintaining traceability relations

Stage 3 of our approach brings the set of traceability relations back into an accurate state by performing traceability updates related to the development activity undertaken. During the recognition of development activities, we are careful to pinpoint those changes that are purely atomic (i.e., the addition of a totally new element, the deletion of an element, or simple modification that does not destroy, enhance or move an element). To differentiate these from wider activities in which they may play a role, we assume that a developer will complete a composite activity within a number of elementary changes and so introduce the concept of delay. This is discussed in more detail in Section 3.4.

If an element is deleted, it can mean that this part of the model is not needed anymore, in which case it is necessary to delete the associated traceability relation(s). The developer will be prompted to confirm this action and the previous state will be versioned. It is equally possible that this deletion is part of a wider activity resulting in new elements. In this case, it would be necessary to transfer the traceability relation(s) from the original element to the resulting element(s) to keep the traceability set accurate. Likewise, if an addition cannot be related to any meaningful wider activity, the enhancement of the model by a new element is assumed. The element is highlighted and tagged because human input is still required when elements are originated. If a wider activity has been recognized and the impacted element was related by traceability relations, these relations can be maintained after the completion of the activity. For elements that evolve as part of a development activity, we maintain the traceability relations in two ways. First, all disconnected relations of an evolving element will be reconnected to the evolved element or composite of elements after the change. Second, if during the modification of an element its replacement or one of its enhancements becomes part of a new parent element, the developer

has the option to move or copy each of the traceability relations on the *old* parent to the *new* parent.

To support change propagation, we distinguish the traceability relations of an element into two groups, incoming relations from dependent elements and outgoing relations to independent elements. To minimize the manual effort, we propagate change only to dependent elements, because we assume a change to a dependent element will not impact the independent one (forward engineering). For example, if a design class is being modified, the change would be propagated to dependent test cases validating the class, but not to the independent requirements defining the class. When change is propagated, all traceability relations of a changing element incoming from dependent objects receive the status *suspect* and are tagged with the description of the recognized change. By this mechanism, we propagate change to all dependent elements and their models, and give support to manually resolve a possible inconsistency. If the dependent object belongs to a model we support (currently only structural UML models) our approach can also maintain the traceability relations of the dependent element while resolving this.

3 Traceability maintenance rules

The approach is built upon rules that allow for the recognition of development activities within a flow of elementary change events. These rules provide a directive to update traceability relations in predefined ways. In this section, the definition of these rules, their representation and their application is presented.

3.1 Rule definition

The viability of our approach depends upon the ability to catalog and capture all developmental activities that have traceability implications for the artifacts we handle. The activities we focus upon are those that recur for most software development practices based upon UML modeling, though obviously depend upon the particular process used, the capabilities of the target programming language and the intended domain.

To catalog representative development activities, we studied several methodologies as well as practice on industrial projects and collected *traceability relevant* activities that typically occur during the analysis and design of systems or due to later changes. Forward engineering processes we consulted included the Unified Process [13] and we support the development activities that Arlow suggests for refining an analysis model into a design model [4]. Also, we consulted Fusion [7], Quasar [18], the V-Model [17] and Refactoring [10]. We obtained a list of 38 development activities with impact on traceability (12 apply to associations, 6 to inheritance, 3 to attributes, 1 to methods, 6 to classes, 5 to components and 5 to packages). Development activities that apply to relations include: refining an unspecified association into one or two directed associations (as per Figure 1); refining an association to aggregation or composition; resolving one to many associations; resolving many to many associations; and resolving association classes. Development activities that apply to classifiers include: moving an attribute, method, class or package; splitting a class, component or package; merging a class, component or package; converting a class to a component; and converting an attribute to a class (see Figure 4).

All the activities identified to date have been captured by 21 rules with 67 alternative ways of occurring. Some are captured by more than one rule (e.g., moving an attribute) and some are only traceability relevant if the impacted model element is being deleted and a new element created, instead of modifying the existent element (e.g., refining association to aggregation).

3.2 Rule representation

Rules have been defined to recognize development activities and are stored in the open XML format. These capture all valid sequences of changes that could comprise the activity as well as information about the necessary update to traceability relations.

The syntax of a rule is given by a self-defined XML Schema Definition (XSD). Each rule is focused on one development activity performed on one type of model element. The head of a rule consists of a distinct <Rule ID>, a description of the development activity it is able to recognize and the type of the model element the activity focuses upon. The rule then consists of one or more <Alternative> sections. These sections reflect different sequences of change events to accomplish the same activity (see Sections 2.2 and 3.3). Each alternative is composed of a <ChangeSequence> and a <LinkUpdate>. The <ChangeSequence> section consists of a definition of all expected elementary change events to recognize the sequence. The <LinkUpdate> section defines all source elements impacted by the activity and all target elements that have been impacted, created or modified during the activity and require update of traceability relations (see Section 3.5).

3.3 Definition of change sequences

A special notation is used to define so-called event masks that are compared with incoming change events

to recognize them. These masks are similar to an event, but include enhanced options to compare event properties. Every event mask has a unique ID which is used to reference properties of other events during mask definition. There is one special trigger event ID="T" within each sequence that allows the rule engine to recognize the development activity (see also Section 3.4). The remaining events are numbered starting with ID="1". To be able to recognize several elementary changes as one activity, it is necessary that the element is being modified or deleted, so the trigger event reflects that elementary change that is modifying or destroying the impacted model element. Only an incoming trigger event allows the recognition of a new development activity. To find related change events belonging to one activity, the properties of the trigger event will be compared with those of the surrounding events.

3.4 Rule application

Figure 2 illustrates the rule application process. On the arrival of a new change event, the rule engine performs the following three tasks:

1. The event is put in an *EventCache* (a first in, first out buffer for the last n incoming events). If full, the oldest event will be deleted, along with its occurrence within all *OpenRules* (rules to which at least the trigger event has been recognized, but with events missing to complete it).
2. The event is assigned to all *OpenRules* with a missing event equivalent to the incoming event.
3. The *RuleCatalog* (predefined rules) is searched for rules with *TriggerEvents* (one characteristic event within each alternative sequence of change events) matching the type and properties of the incoming event. All matching rules are established as new *OpenRules* and the *EventCache* is searched for matching events to complete the *OpenRules*.

If during task two or three an *OpenRule* is completed, its events will be deleted from all other *OpenRules* and will be disabled within the *EventCache*. Subsequently, the predefined traceability update for the elements defined within the rule is performed. This action restores the traceability. For most of the development activities, it is possible to do this in a fully automated manner. Where it is not, we show a dialogue box within the CASE tool to let the developer choose the correct alternative. Note that in rare situations one event contributes simultaneously to many development activities. This is being addressed in ongoing work.

To handle elementary change events that do not impact traceability, as well as unfinished development activities and the addition and deletion of elements that

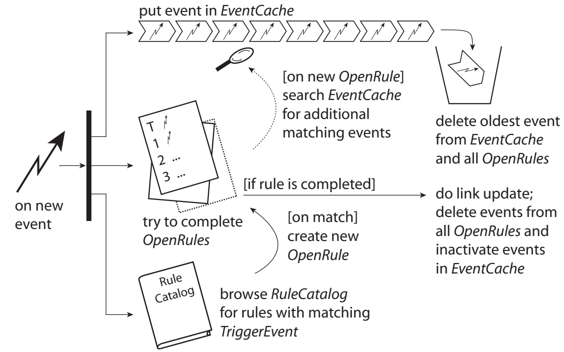

Figure 2. Rule application process

are not part of a wider development activity, the *EventCache* has a configurable size. This mechanism reflects the assumption that the developer would complete an activity within a certain time frame. The development activities we identified comprise a minimum of 2, a maximum of 6 and an average of 3.54 events, so for the evaluation in Section 5 we used a delay (*EventCache* size) of 30 events. This allows a developer to undertake several activities in parallel while mitigating the risk of recognizing activities incorrectly.

3.5 Update of traceability relations

After a development activity has been recognized as completed, the traceability update incorporates two parts: the update of the relations on the element itself and the update of the relations on the parent object of the element. If a user interaction is necessary during the update, we provide a detailed description about the recognized activity and the situation the user has to resolve. In any other situation the update will be carried out automatically in the background.

To update an element's own relations, all links on all update sources (defined within the rule) will be collected and, if not already existent, re-established on all update targets (defined within the rule). We could extract possible update sources and targets automatically from the defined change sequence and save the definition of these, but we found it more convenient to be able to customize the update for each rule alternative.

For the update of the relations on the parent of the impacted element, we compare the parent of the source with all parents of the targets. For any two different parents, we let the developer decide which relations on the source parent will be copied or moved to the update target. These situations occur, for example, if a class is being moved from one package to a different package and these packages are related by different links.

3.6 Status of the rule set

We developed an initial set of rules for working with structural UML elements based on the development approaches mentioned in Section 3.1. This set of rules has been subject to test and now delivers good results, by which we mean they are able to recognize the activities of developers in our studies with high accuracy and perform the required traceability maintenance (see Section 5). The rule set provides for a good starting point and extension is the subject of our ongoing investigations.

4 traceMaintainer prototype

To evaluate our approach, we have developed a prototype. This has been implemented in Visual Studio .Net and uses the Microsoft XML Parser. It supports the following activities: (a) the analysis of a flow of elementary change events according to a set of predefined rules it imports from an XML file; (b) the specification of new rules; and (c) based on a match between events and rules, it restores traceability. The prototype applies rules defined using XML according to our XML Schema Definition (XSD). The use of open and standardized techniques to define our rules ensure their readability by humans as well as provide a simple and small engine for their interpretation and the generation of update directives for the traceability relations.

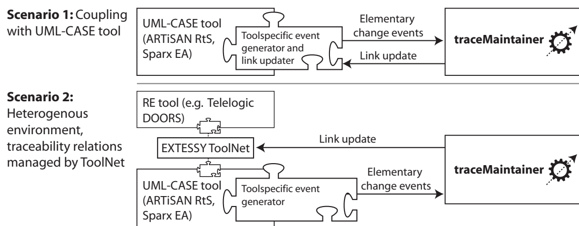

Figure 3. Two possible tool usage scenarios

The prototype has been designed to be independent of specific CASE tools (see Figure 3). The intention is for it to be deployable with every UML-CASE tool that allows for capturing the necessary change events to model elements and that allows the manipulation of traceability relations from outside the tool. It is only necessary to write an adapter for each tool that is to be connected to our prototype, so the prototype basically augments the existing functionality. We have developed adapters to ARTiSAN Studio and to Sparx Enterprise Architect. The main purpose of these adapters is the generation of events and the collection of element properties in order to provide the rule engine with standardized elementary change events (see Section 2.1).

The adapters are also used to allow the rule engine to update traceability relations kept within the development tool. Furthermore, it is possible to use the prototype in heterogeneous settings of requirements and software engineering tools. In these settings, the software development tool is used to capture the necessary change events. The directives for the necessary traceability updates are sent to a different tool, like EXTESSY ToolNet [9], that holds the traceability information. We developed an adapter to ToolNet and use it to hold all our traceability relations even for projects with model elements within only one tool. The reason is its ability to link every element of a model.

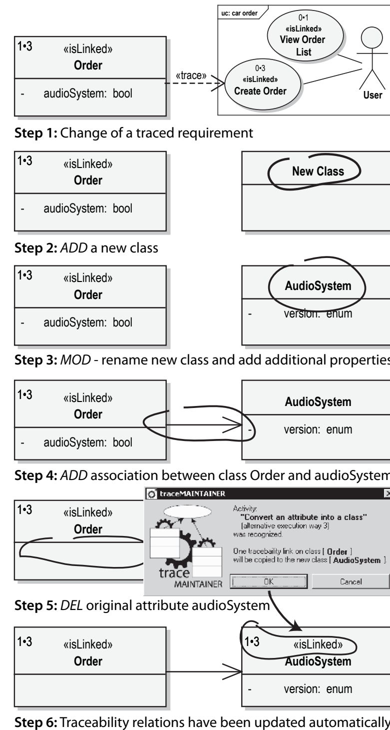

Figure 4. traceMaintainer updates traceability relations as a user converts attribute to class

Figure 4 depicts a usage scenario for traceMaintainer. In the scenario, the analysis model of a car order system has to be changed because of a change request relating to a use case. Step 1 shows the initial situation. The use case *Create Order* is realized by a class *Order*. To show the link between both model elements, a traceability relation exists between them. The class has an attribute *audioSystem*. The customer requests a change of the use case *Create Order* to support different audio system options during the creation of an order. This change in the use case requires a structural change to the system's UML analysis model, which in

turn requires the maintenance of its traceability relations. The attribute *audioSystem* has to be converted into a class. Step 2 to Step 5 show how the developer could carry out the required change. A class *New Class* is created and renamed *AudioSystem*. An association between class *Order* and class *AudioSystem* is created. As a last step, the original attribute *audioSystem* is deleted. traceMaintainer recognizes the completion of the development activity and shows the depicted dialogue before it updates the traceability relation that has been impacted automatically. For this example, update means copying the existing relation on class *Order* to the new class *AudioSystem* as both classes now fulfill the use case *Create Order*. Step 6 shows the newly created traceability relation on class *AudioSystem*. Within the scenario, only one way to convert the attribute into a class has been discussed, though traceMaintainer can handle different ways to accomplish an activity if pre-specified in its rule set.

5 Preliminary validation

We performed two experiments using the prototype to explore the following research questions:

1. Is the approach capable of maintaining traceability relations at a level of accuracy comparable to manual maintenance during the evolution and refinement of UML models?
2. How much manual effort can be saved by using automated maintenance and how dependent is that saving on the kind of modeling undertaken?

The goal of the first experiment was to evaluate the completeness and correctness of a project's set of traceability relations after refining UML models. We used the analysis models of two software systems. The first was abstracted from a wiper control system for a car, created for Volkswagen AG, and the second was a library management system developed by students of the Technical University of Ilmenau. Sparx Enterprise Architect was used as the CASE tool and the traceability relations were managed by EXTESSY ToolNet.

The documentation for the wiper system included: a document with 26 requirements statements related to a UML analysis model; an initial UML analysis class model composed of 21 classes, 35 attributes, 23 associations/generalizations and 28 methods; and an initial set of 105 traceability relations. The documentation for the library system included: a requirements model with 21 use cases related to a UML analysis model; an initial UML analysis class model with 17 classes, 20 attributes, 27 associations/generalizations and 18 methods; and an initial set of 59 traceability relations.

The models for both systems were given to two developers (not authors of this paper). Developer A had 4 years of industrial experience in model-based, object-oriented software development and developer B had 2 years of experience. Both had university-level education on the topic. The task for each developer was to refine the analysis model of the system into a design model that could be implemented. Each developer spent 2 hours on the task and the status of the traceability relations were maintained by traceMaintainer behind the scenes. After finishing the task, the developers manually updated the traceability for the just created design model to the requirements model, in consultation with a version of the initial relations between the requirements and analysis model. This second task took 1 hour. Each developer undertook this task for the wiper control system and then for the library management system.

Table 1. Traceability relations after task execution in experiment 1

	Number of links			Pre-cision	Re-call	tM wrong
	Dev	tM	D∩tM			
Wiper						
Dev A	147	149	140	0.94	0.95	0.16
Dev B	138	132	128	0.97	0.93	0.20
Library						
Dev A	94	92	88	0.96	0.94	0.13
Dev B	98	99	96	0.97	0.98	0.70

The number of traceability relations existing after the completion of the developers' activities for each system are depicted in Table 1 in column Dev. These figures were compared with those provided automatically by traceMaintainer in column tM. The number of correct relations in the tM set (i.e., those also identified by the developer) are listed in the column D ∩ tM. Precision is a percentage measure for the number of relations that have been correctly maintained by traceMaintainer in relation to all the relations existing after the automated maintenance – or: $truePositives/(truePositives + falsePositives)$. Recall is a percentage measure for the number of relations that have been correctly maintained by traceMaintainer in relation to all correct relations – or: $truePositives/(truePositives + falseNegatives)$. Column tM wrong gives the percentage of missing or wrong changes in relation to all the changes performed by traceMaintainer. The figure of up to 20% was found to be due to incorrect or missing rules, illustrating that the rule set needs to be improved iteratively. From further experiments, this set is now converging.

Although provisional, this provides encouraging evidence about the ability of our approach to automate the maintenance of traceability relations at accuracy levels approaching that attained via manual effort. Given that 50% of the task time was spent restoring traceability, this is a considerable saving in effort. A set of 32 rules applied to 7 different types of UML model element were used during this experiment.

To answer our second research question in more detail, we used one part of the refinement activities of developer A on the library system. We analyzed the changes the developer had applied to the model and assembled them into a detailed flow of changes. We then defined two possible paths of execution in terms of elementary changes to accomplish the same overarching development activities. These each have differing impact on and requirements for the maintenance of the associated traceability relations. To compare the manual effort that can be saved by using traceMaintainer it is necessary to examine these differences.

To explore the greatest differences, we defined and executed the scenario in the most optimistic way with the least impact on traceability and in the most pessimistic way with the maximum impact on traceability. Table 2 gives some statistics on the elements and traceability relations of the model for the scenario independent of the execution path.

Table 2. Number of elements and relations within the model of experiment 2

	Before scenario		After scenario	
	Count	Links	Count	Links
Classes	23	67	35	86
Associations	30	25	47	43
Attributes	63	0	66	0
Methods	27	0	27	0

After executing the scenario for the first time and encountering similar results in terms of recall and precision as those in Table 1, we analyzed all the issues and were able to adjust our rules accordingly. We were then able to maintain the relations in the same way they have been maintained manually by developer A.

Dependent on the execution path chosen, either 127 or 176 elementary change events occurred taking 65 and 82 minutes respectively to perform manually (see Table 3). These changes belong to 35 and 49 *traceability relevant* development activities respectively. The surprising result was that the necessary time for the manual maintenance of relations ranged from 62 to 116 minutes after the modeling period. The results show how immense the effort required for manual traceability maintenance actually is and how much it depends on how a modeling activity is executed. A developer's design freedom is possibly lost with awareness of the future work a development strategy choice is likely to cost them. By using traceMaintainer within the same scenario the developers' time required for traceability maintenance could be reduced by 71% to 84%. It consists of user decisions on unclear link updates and additional links created on new elements during design.

After seeing how much effort is necessary to keep traceability in order even for a relatively small scenario, one could question the necessity to maintain traceability at all. To demonstrate what would have happened if the developer had not maintained the traceability relations, we give the Recall and Precision metrics for both paths of execution without traceability maintenance. For the optimistic path, we computed 65% Recall and 91% Precision after all changes and no maintenance. For the pessimistic path, we computed 39% Recall and 54% Precision. The metrics show that, independent of the way a development scenario is undertaken, the traceability relations within the model erode and maintenance is highly desirable for future viability and use.

The results from these experiments are preliminary and there are threats to validity. Given a small set of developers, their tracing activities may not be representative of a wider population. They could have produced poor sets of relations which were not a suitable baseline for comparison with those of the approach. The researchers also discussed traceability with the developers to come to an agreement about traceable elements and when to forge traceability relations, and both traceMaintainer and the <LinkUpdate> section of the rules were customized to reflect this agreement.

6 Related work

Spanoudakis et al. [19] present a rule-based approach for the automatic generation of traceability relations between documents which specify requirement statements and use cases (in structured natural language) and analysis object models. Requirement-to-object-model rules, and a technique based on information retrieval, are used to establish traces. The second kind of rule analyzes the relations between requirements and object models to recognize intra-requirements dependencies and establishes relations. The approach requires the export of all supported artifacts into XML 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 cur-

Table 3. Results of experiment 2

	Changes	Activities	Modeling	Manual maintenance	traceMaintainer	Saved with tM
Optimistic execution	127	35	64,5 min	62,0 min	18,2 min	71%
Pessimistic execution	176	49	82,2 min	115,8 min	18,2 min	84%

rent form, does not appear to support the maintenance of traceability relations following artifact evolution.

Cleland-Huang et al. [5] present an approach called event-based traceability. The authors link requirements and other artifacts of the development process through publish-subscribe relationships. Changes to requirements are categorized by seven kinds and events are raised according to kind. These events are published to an event server that sends notifications about change to subscribers of a changed requirement. The notification contains information to support the update process of the dependent artifacts to facilitate manual maintenance. The approach also maintains traceability between requirements after predefined changes to requirements [6]. There are similarities between the approach and that proposed in this paper. The authors also capture changes to a model (requirements) as events, their model contains one type of element (requirement) with properties of interest, they identify seven possible change activities to requirements and they deal with compound change events by tracking lower level tasks. The authors do not discuss how to recognize the elementary change actions and how to relate them to a compound activity in depth though. For changes to complex models created using UML, the recognition of change becomes crucial, as described and more specifically addressed within this paper.

Olsson and Grundy [15] describe an approach in which they extract key information from different artifacts (requirements specifications, use cases and tests) into abstracted representational models. The developer can then create explicit relations between the abstract elements. Some implicit relations can be defined automatically (e.g., consistently named users within different artifacts). Through this mechanism, changes can be propagated. Some changes can be resolved automatically (e.g., changing the name of a user). For others, developers are informed so they can take appropriate action. In comparison to the work of Olsson and Grundy, we also propagate the change of a traced model element. Additionally, we are able to maintain the traceability relations of evolving model elements in some cases automatically and in others with limited user interaction. In contrast, we do not need to extract the data from the models first and provide the propagated information about change within the model.

Traceability is also supported by many commercial requirements management tools, enabling the tracing of requirements to other artifacts in the software development life cycle. One example, IBM's RequisitePro, allows developers to relate requirements kept within the tool to other tools in the product suite, such as Rational Software Modeler. While these tools support UML explicitly, there is limited support for the automated creation or maintenance of traceability relations at fine-grain levels. To integrate the approach of this paper, it would be necessary to write a tool-specific adapter to generate the necessary events, and to be able to create and delete the traceability relations.

There are a number of ways to support a developer in terms of traceability maintenance, from guiding them through a predefined set of permissible activities that result in traceability to attempting to recognize actions and their implications. Our approach is based upon the latter strategy. The problem of developers doing surprising and unanticipated things may result and incur need for manual intervention, but we assume this is more desirable than removing a developer's freedom to create. The main contribution of our approach is, therefore, that it addresses the lack of effective support for the automated maintenance of traceability relations in UML-based development tools as a by-product of work carried out within them. It uses a rule-based approach to do this based on the modeling of development tasks. The motivation is unique in the desire to *sustain* initial investment in a project's traceability.

7 Conclusions and future work

In this paper, we have presented an approach that supports the automatic maintenance of traceability relations between requirements, analysis and design models of software systems expressed in UML. The approach analyses elementary change events generated while working within a CASE tool. Within the captured flow, sequences of events are sought that correspond to predefined rules. These rules represent various ways in which recurring development activities can be undertaken. Once a sequence has been recognized, the corresponding rule gives directives to update impacted traceability relations and, by these actions, restores the set back to an accurate state. We achieved encouraging results in two early experiments. Several issues were recognized during the experiments which

have led to an improved version of the approach.

Limitations of the approach are that only predefined activities can be recognized at present and these are unlikely to reflect all possible development approaches, so it might be necessary to customize the rules to project specifics. Whereas there is an initial cost in identifying development activities and formulating rules, the rule set is proving reusable within this restricted scope of UML-based object-oriented software engineering. It is a future exercise to gain more statistical data on the cost/benefit trade-off, costs in terms of initially defining the rules and benefits in terms of the time saved on manual maintenance across all projects using the rules.

The findings from the initial experiments highlighted future directions for this work and are informing a planned industrial case study. More empirical studies are needed to evaluate the effectiveness of the rule matching. Addressing the situation where events may participate in many rules simultaneously in an interleaved manner is a research issue and it would be desirable to extend the support to other kinds of development model. The necessary preconditions would be models with a limited number of discrete elements and sufficient properties to be able to recognize meaningful development activities. This requires a stronger relation between traceMaintainer and the development project's traceability meta-model to be able to customize the artifacts that can be related to each other. There may also be some scope to support the definition of rules semi-automatically and to use the rules for checking the consistency of change activities.

Preliminary results show that the approach described in this paper is capable of reducing the effort (and so cost) in maintaining traceability quite dramatically and at a high level of precision. The approach is intended as a complement to those approaches that initially create a set of traceability relations using automated techniques. Further, given the attention that a number of these automated techniques place on establishing traceability between design and code, the approach can potentially fulfill an intermediary role in sustaining the overall traceability picture.

Acknowledgments This work is partly funded by Deutsche Forschungsgemeinschaft (DFG) id Ph49/7-1. The authors would like to thank Tobias Kuschke and Christian Kittler for implementing the prototype.

References

[1] I. Alexander. Toward automatic traceability in industrial practice. In *Proc. 1st Int'l Workshop on Traceability in Emerging Forms of Software Engineering*, pages 26–31, Edinburgh, UK, Sept. 2001.

[2] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE TSE*, 28(10):970–983, 2002.

[3] P. Arkley, P. Mason, and S. Riddle. Position paper: enabling traceability. In *Proc. 1st Int'l Workshop on Traceability in Emerging Forms of Software Engineering*, pages 61–65, Edinburgh, UK, Sept. 2001.

[4] J. Arlow and I. Neustadt. *UML 2 and the Unified Process Second Edition: Practical Object-Oriented Analysis and Design*. Addison-Wesley, 2005.

[5] J. Cleland-Huang, C. K. Chang, and M. J. Christensen. Event-based traceability for managing evolutionary change. *IEEE TSE*, 29(9):796–810, 2003.

[6] J. Cleland-Huang, C. K. Chang, and Y. Ge. Supporting event based traceability through high-level recognition of change events. In *COMPSAC*, pages 595–602. IEEE Computer Society, 2002.

[7] D. Coleman. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1994.

[8] A. Dekhtyar, J. Hayes, S. Sundaram, A. Holdbrook, and O. Dekhtyar. Two heads are better than one, or too many cooks in the kitchen? Technique integration for requirements assessment. In *Proc. 15th Int'l Requirements Eng. Conf.*, pages 69–83, Oct. 2007.

[9] Extessy AG. ToolNET. `www.extessy.com`.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[11] O. C. Z. Gotel and A. C. W. Finkelstein. An analysis of the requirements traceability problem. In *First International Conference on Requirements Engineering (ICRE'94)*, pages 94–101. IEEE CS Press, 1994.

[12] J. H. Hayes, A. Dekhtyar, and J. Osborne. Improving requirements tracing via information retrieval. In *RE*, page 138. IEEE Computer Society, 2003.

[13] I. Jacobson, J. Rumbaugh, and G. Booch. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.

[14] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proc. of the 25th Int'l Conf. on Software Eng. (ICSE-03)*, pages 125–137, Piscataway, NJ, May 3–10 2003. IEEE CS.

[15] T. Olsson and J. Grundy. Supporting traceability and inconsistency management between software artefacts. In *Int'l Conf. Software Eng. and Appl.*, Nov. 2002.

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

[17] V. Schuppan and W. Rußwurm. A CMM-based evaluation of the V-model 97. In R. Conradi, editor, *EWSPT*, volume 1780 of *LNCS*, pages 69–83, 2000.

[18] J. Siedersleben. *Moderne Software-Architektur*. Dpunkt Verlag, August 2004.

[19] G. Spanoudakis, A. Zisman, E. Pérez-Miñana, and P. Krause. Rule-based generation of requirements traceability relations. *JSS*, 72(2):105–127, 2004.