Enabling Automated Traceability Maintenance Through the Upkeep of 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. Traceability is demanded within mature development processes and offers a wide range of advantages. Nevertheless, there are deterrents to establishing traceability: it can be painstaking to achieve initially and then subject to almost instantaneous decay. To be effective, this is clearly an investment that should be retained. We therefore focus on reducing the manual effort incurred in performing traceability maintenance tasks. We propose an approach to recognize those changes to structural UML models that impact existing traceability relations and, based upon this knowledge, we provide a mix of automated and semi-automated strategies to update these relations. This paper provides technical details on the update process; it builds upon a previous publication that details how triggers for these updates can be recognized in an automated manner. The overall approach is supported by a prototype tool and empirical results on the effectiveness of tool-supported traceability maintenance are provided.

Keywords: Automated traceability maintenance; Model-driven engineering; Change management; Rule-based traceability; Traceability update.

1 Introduction

Establishing traceability on a project can be time consuming. Even with the support of emerging automated techniques there is still substantive manual work involved [1]. Providing for traceability is intended to support change management and many other software development tasks but, to leverage this investment, it is necessary to have an accurate and ready to use set of traceability relations. Traceability needs to be maintained while project artifacts are evolving.

By traceability maintenance, we refer to the modification and/or enhancement of existing traceability relations after changes to artifacts to ensure their continued correctness and accuracy. We highlighted a general approach for automated traceability maintenance in a previous paper [2]. The approach targets model-driven software development using UML and comprises two key tasks:

(i) recognizing changes to models in terms of the broader development activity being undertaken; and then (ii) updating the impacted traceability relations to restore the traceability. The technical details underlying the automated recognition of development activities have been described in an earlier paper [3]. In this companion paper, we focus on the technical details associated with traceability upkeep and explain how the necessary updates are implemented.

The paper is organized as follows. In Section 2, we provide high-level details about our approach and describe the context of software development for which it has been developed initially. In Section 3, we describe our process for updating existing traceability relations after the recognition of development activities. We also describe how necessary updates can be propagated. In Section 4, we discuss our initial validation through an experiment that compares the effort and quality associated with tool-supported traceability maintenance, based upon the approach described, versus manual efforts. We end the paper with a summary of related and future work.

2 Motivation and Scope

Within this section, we discuss the problem of maintaining traceability relations and present an overview of our solution to this typically manual task. We first outline the context that we work within.

2.1 Model-driven Software Development

With model-driven software development using the UML, a variable number of abstraction layers (i.e., models) can be created to document a problem and its solution, from the initial requirements through to the final implementation [4]. As the elements of these models describe the same system, there are benefits in establishing explicit traceability relations between models to handle change. Few industrial projects implement traceability in this fashion though, due to the perceived and actual costs, and struggle to find the right balance as project artifacts evolve [5]. Traceability maintenance can be a time consuming proposition.

2.2 Approach and Tool Support

Our approach is founded upon the following assumptions: (1) while evolving any kind of UML model, it is possible to capture the elementary change actions and salient information regarding the properties of the changed element; (2) one can understand the intention of these elementary change actions within the context of a chain of related change actions on an element and so determine the wider development activity; and (3) knowledge of an intentional development activity provides the information necessary for pre-existing traceability relations to be updated following the changes. Our approach therefore records all the changes to model elements and uses this information to find a match within a set of

predefined patterns of recurring development activities. A match will instigate the requisite traceability update actions.

Changes to a UML model can be classified into three elementary types: adding new elements, deleting elements and modifying existing elements. All elementary changes that are not recognized as part of a wider development activity are handled as new additions or deleted elements of the model. For new elements, we support the developer in the creation of traceability relations. For deleted elements, we discard associated traceability relations if necessary. To restore overall traceability within a set of interrelated models, we support the propagation of required changes to models.

Currently, the approach is restricted to the analysis of changes to those models described via structural UML diagrams (e.g., class, object, composite structure, package and component diagrams). We focus on UML models as these are the de facto standard for model-based software development and a wide range of existing CASE tool support is provided. Furthermore, UML offers the possibility of capturing the full range of software development artifact via its diagrams and offers basic support for traceability relations as part of its meta-model [6]. We are currently investigating how to extend the approach to support the behavioral diagrams of the UML and also to handle additional types of model. More details are described in earlier papers ([2], [3]) and the approach is supported by a prototype tool called *traceMAINTAINER* [7].

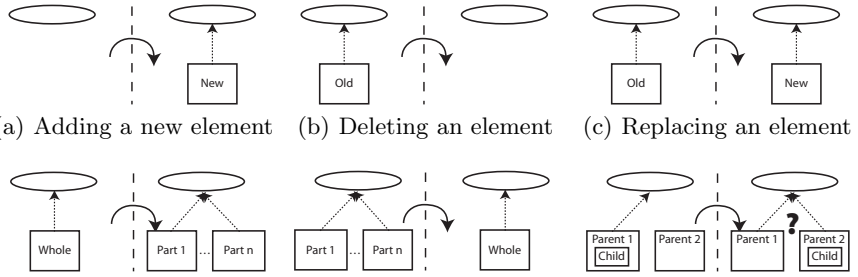
3 Traceability Update Process

We restrict our focus to post-requirements traceability [8] and hence to the consequent modeling activities and artifacts of the software development lifecycle. The development process therefore consists of activities that incrementally transform a requirements specification into the final implementation (e.g., the platform specific model or source code) in a forward engineering manner. Each of these activities is applied to or influenced by various input artifacts and creates new or improved output artifacts. Creating an explicit traceability relation between these artifacts can capture these dependencies. We represent our traceability relations as stereotyped dependency relationships, as defined by the UML meta-model. The direction of a dependency relation points, by default, from the dependent model element towards the independent model element. This directionality is intended to convey semantics and, in our case, to assist traceability update and change propagation, but does not prevent bi-directional use or navigation of the traceability relation itself.

3.1 Types of Model Changes

Focusing on post-requirements traceability relations and restricting directionality, requirements can be treated as sinks of a directed acyclic graph, sourced by elements of implementation, test and so on. The nodes of this graph represent related elements in different models and the arcs represent traceability

relations. Each related requirement spans such a tree. Changes to models that impact traceability are the possible changes to the traceability graph and are combinations of the elementary changes given in Section 2.2. In this section, we describe the types of change we distinguish at the model level and explain their relation to the change of the traceability graph. Figure 1 depicts all the change types and their traceability implications. Figure 2 depicts the underlying graph that is the subject of the examples in the sub-sections below.



(a) Adding a new element (b) Deleting an element (c) Replacing an element

(d) Splitting into several parts (e) Merging into one whole (f) Moving into another parent element

Fig. 1. Different change types with a focus on the necessary traceability updates

Adding a New Element This change enhances a model by adding a new element (see Figure 1(a)). While creating new traceability relations on the element, new nodes will be added to the traceability graphs of the associated requirements.

Recognition of change: Each change to a model is retained in a buffer of our *traceMAINTAINER* tool until the wider development activity has been identified. If we have not been able to assign the creation of a new model element (i.e., ADD event) to a wider development activity, we treat the change as an enhancement of the model. This means that the element is a new additional part of the model and may require the creation of a new traceability relation. The necessity to establish new traceability relations depends upon the project's traceability information model.

Required traceability update: After adding a new element, it might be necessary to relate the element to independent elements that are the reason for the addition. The traceability information model specifies what elements are allowed/intended to be linked for a project, so it is possible to suggest or even require the creation of one or more traceability relations on the new element. In cases of ambiguity, we add a tag to such elements for the developer to examine when convenient. On exiting a CASE tool, a dialog is provided to alert the user to any tagged elements that are yet to be connected within the traceability graph. The dialog provides possible counterparts for new traceability relations

on the element according to the traceability information model. Also, the last traced element of that type will be offered, as it is common that several proximal changes belong to the implementation of the same new requirement. Recently changed elements are more highly ranked and we are currently investigating the possibility of using existing information retrieval techniques to provide a better ranking of candidate relations to aid this task.

Impact on existing traceability: This change type has no impact on the existing traceability relations but, if new elements are not related, the results of impact analysis and change propagation are likely to be wrong.

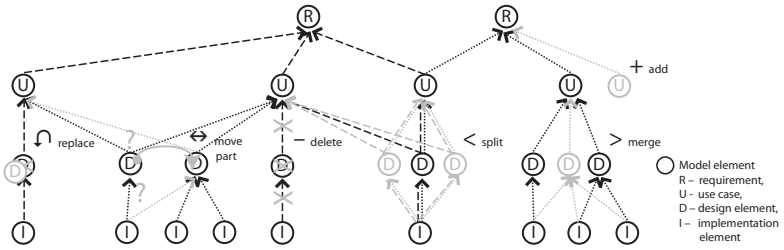


Fig. 2. Example of two overlapping traceability graphs implementing two requirements

Deleting an Element This change arises when an element is removed from a model without being replaced (see Figure 1(b)). While removing an element, its relations will also be deleted, so nodes from the traceability graphs are removed.

Recognition of change: This is similar to ADD events. For each autonomous DEL event that is removed from the *traceMAINTAINER* buffer we assume that the intention behind the activity is to remove the element from the model.

Required traceability update: If an element has been removed it is necessary to remove its traceability relations and examine the resulting impact. Using *traceMAINTAINER*, we store the traceability relations in an additional repository with the advantage that we are able to find out about an element's relations even after its deletion, but with the necessity to delete these inconsistent relations explicitly.

Impact on existing traceability: The deletion of inconsistent traceability relations is sufficient for relations to independent elements. For incoming links from dependent elements, however, it is necessary to check whether these elements are still valid and required or not. All these traceability relations will be kept and receive a *suspect* tag and the deletion of the model element will be added as the rationale for this status. Developers need to make decisions on suspect tags.

Replacing an Element An element may be replaced by another element for a number of reasons (see Figure 1(c)). It can be refined or generalized (e.g.,

replacing an association by an aggregation or composition and vice versa, or replacing a class by an interface and vice versa). The element can also be converted into a different type (e.g., converting a class into a component and vice versa). Replacing an element requires restoring all former traceability relations on the new element. This means that the nodes of the traceability graphs have to be replaced.

Recognition of change: The replacement of an element is a wider development activity comprised of several elementary changes. At a minimum, the deletion of the old element and addition of the replacing element are present. We use predefined rules and a rule engine within *traceMAINTAINER* to recognize such development activities, as described in an earlier paper [3].

Required traceability update: In general, the update after a replacement activity requires the restoration of all hanging traceability relations that were related to the impacted model element on the newly created replacing element. Restoration here means to delete all hanging relations of the old element and to recreate them on the new element.

Impact on existing traceability: The replacement element may have an impact on the elements of dependent models requiring an update. Our change propagation mechanism is described in Section 3.2.

Spitting or Merging Elements An element may be split into two or more elements and two or more elements may be merged into one resulting element (see Figures 1(d) and 1(e)). Examples for split and merge are the extraction of a class's attribute into its own class and vice versa, or the refinement of an unspecified association into two unidirectional ones and vice versa. The splitting or merging of elements requires the duplication of existing traceability relations to these elements. This leads also to the split or merge of nodes within the traceability graph.

Recognition of change: As with replacing an element, split and merge are wider development activities comprised of several elementary changes. We provide rules to recognize these development activities within the rule catalog of *traceMAINTAINER*.

Required traceability update: As for the replacement of elements, there are many ways within a typical CASE tool to get to the same result. This variety is important for split activities since traceability update may require copying all the traceability relations that still exist on the modified element to all additional new elements, with or without the removal of the original. The situation is similar for merging. One of the parts might be modified into the resulting whole or all parts deleted prior to creating a new whole. The traceability update requires consolidation of all traceability relations of all the parts on to the resulting whole.

Impact on existing traceability: The impact is similar to replacing an element. The split or merge may have an impact on related elements of dependent models and require change propagation (Section 3.2).

Moving an Element An element or one of its parts may be moved into another context, so into a different element (see Figure 1(f)). Examples would be moving a class into another package or moving an attribute into a different class. Changing the context of an element may require copying or moving the traceability relations on the element's parent to the new parent. This change is similar to adding/deleting a node to the traceability graph.

Recognition of change: As discussed for replacing, splitting and merging, the moving of an element is also a wider development activity and can be recognized by predefined rules within *traceMAINTAINER*.

Required traceability update: It is possible to perform this change type in different ways. The element might be dragged and dropped in a project browser from one element to another. In this case, no update of traceability relations on the element itself is necessary. The element may also be deleted and recreated at the new location. In this case, it is necessary to restore all the traceability relations of the deleted element on the newly created element. The more challenging aspect of the traceability update after moving activities such as these is that of changing the context. It might be the case that the moved element was part of a related element. In such a case, it is likely that these relation(s) are impacted by the move, so we provide a dialog to the developer that shows all the traceability relations on the former parent element (old context) and offers (for each relation) to delete it from the old parent and also to create it on the new parent. This approach offers the developer the possibility of either leaving, copying or moving each traceability relation.

Impact on existing traceability: This change type requires propagation, as per Section 3.2.

3.2 Change Propagation

The main purpose of our work is to maintain the ongoing relevance of traceability relations once they have been established and to reduce the manual effort required to do this. It is for this reason that we distinguish between incoming and outgoing relations on a model element. An outgoing relation points towards an independent element; an incoming relation relates from a dependent element. We use this information specifically for the propagation of changes between different models and/or different levels of abstraction. The related elements in such a context usually have an ancestor/successor type of association. If the ancestor element changes, it is likely that this has an impact on related dependent elements, but it should have no impact on the independent related elements. Rather than make assumptions, the developer may be alerted to re-examine such impact. We therefore propagate changes, by default, only on incoming traceability relations to dependent elements. Nevertheless, the developer may also choose to propagate only to independent elements or, in certain cases, to all related elements. The possiblity to propagate changes also to independent elements can be helpful in situations where a model is changed without changing the more abstract model first so that both models stay aligned.

By propagating changes, we mean that we set all incoming and/or outgoing traceability relations of a changed model element to a *suspect* status and require the developer to manually inspect and, if necessary, resolve inconsistencies. We propagate all changes to an element, even those that did not take part in a wider development activity. This makes sense in a forward engineering process where changes will be propagated to subsequent elements whose creation was influenced by the changed element. To help resolve possible inconsistencies, we capture the recognized change type as a tag on the traceability relation while setting its status to suspect, as mentioned earlier.

We provide a mechanism to resolve relations whose state has been set to suspect. If an element has been modified that has outgoing relations with suspect status then we propose to remove that state after the change. This mechanism reflects our assumption that a developer working on a model element will usually resolve all open issues and ensure consistency to the independent model. Empirical studies are required to examine developers' activities in more detail to ensure the approach is reflective of practical needs.

4 Validation

In this section, we outline an experiment undertaken to explore the following research questions regarding traceability upkeep. Note that validation of the change recognition process has been described elsewhere [3].

Research Question 1 Does use of the *traceMAINTAINER* prototype (a tool that implements the approach and updates traceability relations using automated and semi-automated strategies [7]) reduce the manual effort necessary for maintaining traceability relations? While no manual effort would be a desirable target, we seek evidence of a reduction greater than the time necessary to configure and learn how to interact with *traceMAINTAINER* to make its use worthwhile.

MEASURE: 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 many tasks, so attempting to gain this data would interfere with the task itself. As an indicator of effort, we record the number of performed changes to the set of traceability relations, along with time spent responding to *traceMAINTAINER* dialogs.

Research Question 2 Do the traceability updates performed by *traceMAINTAINER* result in a set of traceability relations of comparable quality to those when maintained purely manually?

MEASURE: Determining the quality of a set of traceability relations depends upon having an agreed baseline. Three types of changes to the traceability relations are then distinguished: changes that have been performed correctly Δ_c (according to the baseline); changes that have been performed incorrectly Δ_i;

and changes that have not been performed Δ_m. To be able to compare the quality and number of changes amongst subjects we compute two measures that are commonly used to evaluate approaches dealing with uncertainty in recognition processes, precision and recall. Precision tells us about the quality of the performed changes, $Q_P = \Delta_c/(\Delta_c + \Delta_i)$ while recall tells us about the number of necessary changes performed, $Q_R = \Delta_c/(\Delta_c + \Delta_m)$.

4.1 Experimental Set-up

Hypothesis Formulation: Our experiment has one independent variable (the use of *traceMAINTAINER*) and two treatments ($tM, no-tM$). It has five dependent variables, on which treatments are compared:

C_m	Number of manually performed changes to the set of traceability relations.
C_a	Number of automated changes to the link-set.
C_{UI}	Number of user interactions with *traceMAINTAINER*.
Q_P	Precision of performed changes (correct changes/(correct+incorrect changes)).
Q_R	Recall of necessary changes (correct changes/(correct+missing changes)).

Null hypotheses:	Manual changes	$H_0 : C_m(tM) \geq C_m(no - tM)$
	Precision	$H_0 : Q_P(tM) = Q_P(no - tM)$
	Recall	$H_0 : Q_R(tM) = Q_R(no - tM)$

Development Project: The experiment was conducted on the UML models for a mail-order system, a completed project implemented in Java by the first author of this paper. The project artifacts include UML models on three levels of abstraction: requirements, design and implementation. The models provide 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 model elements are listed in Table 1(a). The set of traceability relations for this project (referred to as the project link-set) relates the three models and consists of 214 traceability relations. The initial linking was undertaken according to a traceability information model for the project. This states that only relations between requirements/analysis and analysis/design are valid (i.e., no intra model relations and no requirements/design). 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 be related to any other element as long as the rules above are followed.

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 large parts of the system. Although the underlying source code was to be made available within the implementation model, the tasks only required changes to the analysis and design models. It was estimated that it would take 2 to 3 hours 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 are not comparable per se.

Table 1. Development project and subject information

(a) Project models and elements

	RQs	Analysis	Design
Use case diagrams	3		
Class diagrams	1	6	6
Package diagrams		1	1
Activity diagrams	7		
State charts	3		
Sequence diagrams		5	
Package		5	5
Class		41	63
Attribute		73	150
Method		124	280

(b) Prior experience of subjects

	Mean	SD
Programming [years]	11,79	7,58
Languages [count]	4,06	1,70
Projects [count]	2,47	1,29
Projects [days]	448,88	397,20
UML [1-4]	2,80	0,91
CASE tools [1-4]	3,16	0,85
Sparx EA [1-4]	2,81	0,69
Traceability [1-4]	2,05	0,75

(Scale [1-4] – 1 is low and 4 is high)

The tasks comprised: (1) Enhance the system's functionality to distinguish private and business customers and to handle different properties for them. Enhance it also to handle foreign suppliers (including currencies and taxes). (2) Convert two parts of the system into separate components. (3)Enhance the functionality to categorize different products.

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 (tM and $no-tM$), to equally distribute expertise based upon prior experience (see Table 1(b)).

Experimental Procedure:

1. All subjects completed a questionnaire to capture their background and experience. The answers were used to divide the subjects into two groups.
2. All subjects were asked to install the CASE tool to be used (Sparx Enterprise Architect) and to follow a tutorial one week prior to the experiment.
3. All subjects completed a second questionnaire to capture the effort they spent on learning the CASE tool.
4. All subjects spent 30 minutes on a lesson explaining the general structure of the project's UML models and the purpose of the system. The subjects were introduced to the advantages and problems of traceability, the project's traceability information model and how to maintain traceability relations manually within the CASE tool. The subjects of the tM group also received an introduction to the purpose and required responses to requests for user interactions when using *traceMAINTAINER*.
5. All subjects received the description of the three tasks along with a questionnaire that would be used to gather information about the work completed on each task. Only the subjects of the $no-tM$ group were asked to maintain the traceability relations along with undertaking the modeling activities.

6. After 120 minutes, the subjects of the tM group were asked to stop the modeling work and to manually maintain the traceability relations.
7. After 150 minutes, all the subjects stopped work.
8. Each subject participated in a short final interview to gather data on the perceived usefulness of traceability to the tasks, the problems they experienced with traceability maintenance and suggestions they had on improving tool support for traceability.

Data Gathering: Data were gathered via the three questionnaires, as described above. 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 analysis and the participants of the tM group were asked to save their model before the 30 minute manual traceability maintenance period.

4.2 Results

Univariate analyses of the dependent variables were performed to test the hypotheses both individually for each task and across all tasks. For all dependent variables C_m, C_a, C_{UI}, Q_P and Q_R, two-sample t-tests were performed. The level of significance for the hypotheses tests was set to $\alpha = 0.05$. We provide p-values in the t-test columns of Table 2(a) and 2(b). We had to exclude one subject from each group from the analysis, because they did not provide a minimal solution to each modelling task. This precondition was required in order to compare results between all subjects.

4.3 Discussion

Research Question 1 When looking at the number of manual changes C_m to the link-set over the three tasks, the tM group performed far fewer changes (82%) than the $no - tM$ group (see Table 2(a)). This difference is statistically significant. However, it is evident that the $no - tM$ group performed only half as many changes (36.3) than the tM group's combined total of manual and automated changes (6.6+59.6=66.2). There are two reasons for this. First, *traceMAINTAINER* recognizes small incremental change activities and updates traceability relations immediately (in the background) after recognition. This means that the link-set reflects each detour of the developer, in contrast to manual maintenance where the update is typically performed after completing the whole task, resulting in fewer changes. Second, manual maintainers often chose to perform only the minimum required changes to comply with the traceability information model.

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.

Table 2. Descriptive statistics

(a) Change actions and interactions

Task	Var	Treat	Mean	SD	% diff	t-test
All	C_m	no-tM	36.3	12.8	-82%	0.00
		tM	6.6	3.8		
	C_a	tM	59.6	34.7		
	UI	tM	9.0	4.1		
1	C_m	no-tM	18.6	9.5	-87%	0.00
		tM	2.4	1.8		
	C_a	tM	36.2	23.9		
	UI	tM	2.0	2.0		
2	C_m	no-tM	7.7	4.9	-90%	0.01
		tM	0.8	1.8		
	C_a	tM	13.0	3.3		
	UI	tM	6.2	4.9		
3	C_m	no-tM	10.0	4.7	-66%	0.04
		tM	3.4	4.7		
	C_a	tM	12.4	14		
	UI	tM	0.8	0.8		

(b) Precision and recall of changes

Task	Var	Treat	Mean	SD	% diff	t-test
All	Q_P	no-tM	79.5	25.7	21%	0.19
		tM	95.9	4.3		
	Q_R	no-tM	71.3	27.6	11%	0.61
		tM	78.8	19.0		
1	Q_P	no-tM	78.9	36.5	21%	0.34
		tM	95.7	6.0		
	Q_R	no-tM	78.0	36.3	-6%	0.82
		tM	73.3	32.7		
2	Q_P	no-tM	83.3	31.0	19%	0.29
		tM	98.9	2.5		
	Q_R	no-tM	59.5	35.3	51%	0.1
		tM	90.0	14.1		
3	Q_P	no-tM	81.6	37.5	17%	0.44
		tM	95.6	6.5		
	Q_R	no-tM	76.2	35.8	-11%	0.67
		tM	67.8	27.3		

To estimate this indirectly, we counted the manually created relations, manually deleted relations and user interactions. Based upon several measurements with different subjects' data, we correlated the comparable effort of the three direct measures as follows: $T_{create} \approx 2 * T_{delete} \approx 2 * T_{UI}$. Using this, we compared no-tM ($T_{create} + 0.5 * T_{delete}$) and tM ($T_{create} + 0.5 * T_{delete} + 0.5 * T_{UI}$) to gain an approximation of the saved effort by using our approach, as shown in Table 3.

Table 3. Manual effort of the tM group, compared with the $non - tM$ group

Task	All	1	2	3
Approx. effort tM	-71%	-82%	-48%	-67%

Research Question 2 The values of Q_P and Q_R, information about the precision and recall of changes to the link-set (see Table 2(b)), show that the tM group reached a value of over 95% precision for all tasks, with a low standard deviation, 21% higher than the value for the $no - tM$ group. Among all the changes performed by *traceMAINTAINER* we found no incorrect ones; all the incorrect changes within the tM group were manually performed changes. The values for recall are lower than those for precision and, except for Task 2, comparable between both groups. Values of recall lower than 100% indicate that more changes to the link-set would have been necessary. This means that the approach performs updates with high quality (high precision), but does not perform all

the necessary updates. However, any differences between the two groups for both measures are not statistically significant. The quality of the changes performed by both groups is comparable.

4.4 Threats to Validity

External Validity: From a task perspective, the reported experiment is realistic. We had young professionals working on a real project, using commercial tools and implementing demanding tasks. 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 our approach. There are threats associated with the short time the subjects spent on the experiment given the task complexity. However, we did not want to set trivial tasks with obvious changes. A high-level task description enabled there to be 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.

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. Subjects were randomly assigned to the groups to balance expertise, but in order to have comparable results among participants we had to exclude two subjects from the experiment since they did not solve the modeling tasks sufficiently. 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 user interaction for the *tM* group. None of the subjects had any prior knowledge about the approach nor did they know the experimental goals.

Construct Validity: Construct validity refers to having established correct operational measures for the constructs being studied. To investigate the effect of our 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, we used six types of diagram at different levels of detail, our subjects had state-of-the-art education in UML development and we used a widely distributed CASE tool. From the debriefing interview, we learned that almost all the subjects felt immediately familiar with the tool after their prior tutorial. The examination of the resulting models showed that, except for two cases (explained above), all the subjects were able to edit and enhance the UML models in a manner comparable to industrial practice. To investigate the quality of changes to the link-set, the main problem with comparing traceability is the lack of an agreed standard. We therefore provided a traceability information model to give guidance on how to establish traceability for the project. We did the initial creation of traceability relations according to the model and required our subjects to do likewise. In order to gain comparable results, we made further restrictions as to the minimal number and direction of relations (see Section 4.1).

5 Related Work

The goal of our approach is to support the maintenance of already established traceability relations. We are not concerned with creating an initial set of relations, which is mostly the domain of techniques based on information retrieval and data mining, and are concerned more with incremental additions to an evolving set. There is related work on maintaining traceability relations, supporting inconsistency management and change propagation between models.

Maletic et al. [9] describe an XML-based approach to support the evolution of traceability relations between models. The authors describe a traceability graph and its representation in 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. However, the authors do not discuss how to detect these changes in depth nor how to update the impacted traceability relations.

Murta et al. [10] describe an approach called ArchTrace that supports the evolution of traceability relations between architecture and implementation. The use of xADL for the description of architectures and Subversion for the versioning of source code is required. The authors trigger a set of eight policies on committing a new version of an artifact. These policies mostly ensure the update of existing traceability relations on artifacts to new versions within the version control system and further restrict the creation of new relations on old artifacts. The authors do not discuss the recognition of structural changes to supported models nor how to update relations in this case.

Mens et al. [11] 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 to detect and resolve these. They transform the models into a supported format, apply their rules and suggest model refactorings based on the results. While the authors discuss the necessity for traceability management and change propagation while UML models are evolving, they provide no support for this.

Cleland-Huang et al. [12] present an approach called event-based traceability (EBT). 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. Events are published to an event server that sends notifications about the change to subscribers. Change is propagated through sending messages to stakeholders. The notification contains information to support the update process of the dependent artifacts. This facilitates manual maintenance. The EBT approach is similar to our approach, though the authors mainly focus on requirements models and do not discuss how to detect and resolve changes to additional UML models.

Olsson and Grundy [13] describe an approach where 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 action. Like Olsson and Grundy, we also propagate the change of a traced model element and maintain the traceability relations of evolving model elements, in some cases automatically and in others with limited developer 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.

Grundy et al. [14] review existing approaches to handle inconsistencies, between analysis, design and implementation specifications. They also outline requirements for effective inconsistency management and provide exemplars to demonstrate and evaluate their approach. We tried to follow their requirements with our approach, so the necessity to propagate changes immediately as they occur and mechanisms to inform the developer about inconsistencies.

Traceability is 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. We have found this to take from several days to two weeks based on the particular tool.

6 Conclusions and Future Work

This paper addresses the problem of traceability decay by presenting an approach for the maintenance of traceability relations. The approach is currently limited in scope such that it focuses on restoring traceability following changes to related elements within structural UML models while undergoing model-driven software development. The paper provides a set of potential change types and described the necessary update to existing traceability relations that each type demands. The identification of these change types becomes possible by applying development activity recognition rules to elementary change events captured while working within a CASE tool [3] and the approach is supported by a prototype tool called *traceMAINTAINER* [7].

By recognizing each elementary change to a model, it is much easier to solve small incremental problems associated with maintaining traceability. Through our rules and identified change types, this is what we do and we achieve encouraging results. We have conducted preliminary studies to examine the effectiveness of the traceability update process in practice, in terms of effort saved and quality of the end results. We are currently starting a larger longitudinal industrial case study to evaluate our work further. Within that study we plan to gain more statistical data on the cost/benefit trade-off of the approach for practical application.

Acknowledgments This work is part funded by DFG grant Ph49/7-1. The authors thank Johannes Langguth for his work in preparing and analyzing the experiment and all the students who were involved.

References

1. Hayes, J.H., Dekhtyar, A., Sundaram, S.K.: Advancing candidate link generation for requirements tracing: The study of methods. IEEE TSE **32**(1) (2006) 4–19
2. Mäder, P., Gotel, O., Philippow, I.: Rule-based maintenance of post-requirements traceability relations. In: Proc. 16th Int'l Requirements Eng. Conf., Barcelona, Spain (September 2008)
3. Mäder, P., Gotel, O., Philippow, I.: Enabling automated traceability maintenance by recognizing development activities applied to models. In: Proc. 23rd Int'l Conf. on Automated Software Engineering ASE, L'Aquila, Italy (September 2008)
4. Lano, K.: Advanced systems design with Java, UML, and MDA. Elsevier Butterworth-Heinemann, Amsterdam, The Netherlands (2005)
5. Egyed, A., Grünbacher, P., Heindl, M., Biffl, S.: Value-based requirements traceability: Lessons learned. In: Proc. 15th Int'l Req. Eng. Conf. (2007) 115–118
6. Object Management Group Framingham, Massachusetts: OMG Unified Modeling Language Specification (Version 2.1.2). (November 2007)
7. Mäder, P., Gotel, O., Kuschke, T., Philippow, I.: traceMaintainer – Automated Traceability Maintenance. In: Proc. 16th Int'l Requirements Eng. Conf., Barcelona, Spain (September 2008)
8. Gotel, O.C.Z., Finkelstein, A.C.W.: An analysis of the requirements traceability problem. In: First Int'l Conf. on Req. Eng. ICRE, IEEE CS Press (1994) 94–101
9. Maletic, J.I., Collard, M.L., Simoes, B.: An xml based approach to support the evolution of model-to-model traceability links. In: Proc. TEFSE '05, New York, NY, USA, ACM (2005) 67–72
10. Murta, L.G.P., van der Hoek, A., Werner, C.M.L.: Archtrace: Policy-based support for managing evolving architecture-to-implementation traceability links. 21st Int'l Conf. on Automated Software Engineering ASE (Sept. 2006) 135–144
11. Mens, T., van der Straeten, R., Simmonds, J.: A framework for managing consistency of evolving UML models. In Yang, H., ed.: Software Evolution with UML and XML. IGI Publishing, Hershey, PA, USA (2005) 1–30
12. Cleland-Huang, J., Chang, C.K., Christensen, M.J.: Event-based traceability for managing evolutionary change. IEEE TSE **29**(9) (2003) 796–810
13. Olsson, T., Grundy, J.: Supporting traceability and inconsistency management between software artefacts. In: Int'l Conf. Software Eng. and Appl. (Nov. 2002)
14. Grundy, J.C., Hosking, J.G., Mugridge, W.B.: Inconsistency management for multiple-view software development environments. IEEE TSE **24**(11) (1998) 960–981