# Enabling Automated Traceability Maintenance by Recognizing Development Activities Applied to Models

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

*For anything but the simplest of software systems, the ease and costs associated with change management can become critical to the success of a project. Establishing traceability initially can demand questionable effort, but sustaining this traceability as changes occur can be a neglected matter altogether. Without conscious effort, traceability relations become increasingly inaccurate and irrelevant as the artifacts they associate evolve. Based upon the observation that there are finite types of development activity that appear to impact traceability when software development proceeds through the construction and refinement of UML models, we have developed an approach to automate traceability maintenance in such contexts. Within this paper, we describe the technical details behind the recognition of these development activities, a task upon which our automated approach depends, and we discuss how we have validated this aspect of the work to date.*

Keywords: Automated traceability maintenance; Change management; Development activity recognition; Rule-based traceability.

## 1 Introduction and motivation

Traceability serves to provide a logical connection between artifacts of the software development process, at levels of granularity deemed appropriate on a project-by-project basis, so is a mechanism that supports all those activities that require such an overview of a project [6]. For change management, traceability delivers important information about the possible consequences of a changing requirement on other requirements and artifacts of subsequent development stages. For project management, traceability supports the control of a project's progress, as well as provides a way to demonstrate the realization of user requirements. In short, traceability is essential for quality-oriented software development practices.

Though widely accepted as beneficial, the costs associated with traceability can be considerable, so the return on investment is debated [1], [3]. Unless mandated, traceability is rarely extended and used throughout all development stages, due firstly to the number of artifacts or elements therein that would have to be related to yield value, and secondly to the need to maintain these relations each time a change occurs. Even where the set of relations is minimal, the maintenance of this traceability demands time and care. While much attention has been directed toward approaches for establishing traceability initially amongst artifacts, less attention has been paid to ensuring this traceability remains relevant over time. This is the problem of traceability decay and the subject of our work.

The maintenance of traceability relations is a multistep activity. Firstly, as changes occur to the artifacts of software development, it is essential to appreciate both where and how these artifacts play a role with respect the current traceability, along with an understanding of the encompassing development activity that can help to characterize the nature of the change. Secondly, it is necessary to understand the impact of the development activity on the traceability and to carry out those subsequent activities that can re-establish the traceability to at least the prior levels. These core tasks: (1) recognizing those artifact changes that matter; and (2) the updates that are needed to bring the traceability back into balance, should be connected with each other in order to focus on maintaining traceability relations automatically. They demand effective method and tool support to offer a situation in which the benefits of traceability exceed its costs.

We introduced our approach for automated trace-

ability maintenance in a previous paper [10]. This earlier paper provided an overview of the approach and described its focus on automatically maintaining traceability within the context of UML-driven software development. In this current paper, we focus on the technical details associated with the first of the two core tasks, that of recognizing those artifact changes that matter. The technical details underlying traceability update will be the focus of a future paper.

The paper is organized as follows. In Section 2, we provided high-level details about our approach to automated traceability maintenance and the context of software development for which it has been developed initially. In Section 3, we describe how development activities are monitored within a tooling environment and how change events are generated. In Section 4, we explain the rules we use to recognize recurring development activities. In Section 5, we detail the rule application process. In Section 6, we discuss our initial validation, and then we end the paper with a summary of related and future work.

## 2 Automated traceability maintenance

Our approach is founded upon the following observations: while changing any kind of model/artifact, it is possible to capture the rudimentary change actions and information regarding the properties of the changed element; one can understand the intention of these simple change actions within the context of a chain of related change actions on an element comprising a wider development activity; and knowledge of an intentional development activity provides the information necessary for pre-existing traceability relations to be updated to reflect the changes. Our approach therefore records all changes to a model and uses this information to find matches between a set of predefined patterns of development activities, which in turn instigates requisite traceability actions.

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 are currently investigating how to extend the approach to support the behavioral diagrams of the UML and also to handle additional types of model.

Model-based software development offers a way to address the problems of increasing size and complexity of software systems [8]. A variable number of abstraction layers (models), with increasing level of detail, can be created to document a problem and its solution, from the initial requirements through to the final implementation. As the elements of these models describe the same system, there are benefits in establishing explicit traceability relations between these models to handle change. Such development approaches are characterized by iteration, so changes to model elements happen regularly, making traceability maintenance a valuable but time consuming proposition. Few industrial projects implement traceability in this fashion due to perceived and actual costs, as highlighted earlier.

Our approach observes elementary changes applied to UML models, recognizes the broader development activities and triggers the automated update of impacted traceability relations. Figure 1 illustrates the approach and its constituent parts. Within this paper, we only describe the technical details behind the automated recognition of development activities, not the update mechanism. We assume the developer to be using a CASE tool for the creation and change of UML models. A tool-specific event generator provides standardized events to a rule engine (see Section 3). The rule engine (see Section 5) stores a configurable number of change events within a buffer and matches these to predefined rules from a rule catalog. These rules and their representation are explained in Section 4. The recognition of change type therefore happens in the background and in parallel with the developer performing a change on a model within the tool. Once this information is available, it can be used to support the developer in various software development tasks (see Section 5.3). For our research, this means to perform the necessary update of traceability relations, so we are primarily interested in recognizing those model changes which impact this and updating accordingly.

## 3 Generating events for model changes

The structural UML diagrams consist of a limited number of model elements, so we focus on classes, components, packages, attributes, methods, associations, dependencies and inheritance, and all stereotyped versions of these. We distinguish three basic types of change to models described by these diagrams. Elements can either be added, deleted or modified. We refer to such actions as elementary changes.

For each type of elementary change, events are generated. Events contain information about the type of change and properties of the changed element. For the addition of an element, these properties exist only after the creation of the element, and for deletion they exist only before destruction. For modification, both pre and post modification properties are available and required for analysis. We thus define four elementary change events: *ADD*, *DEL*, *preMOD* and *postMOD*.

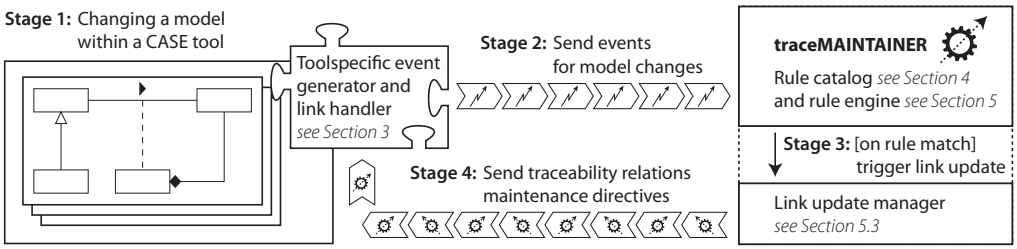


**Figure 1. Overview of the approach and its constituent parts**

These elementary changes form part of wider intentional development activites. To be able to find related changes in the flow of all elementary changes applied to a model, additional properties of a changed model element have to be compared. The minimal required properties of an element are its ID and its type, but other properties are necessary to allow for a distinct recognition depending on the type of element involved. Figure 2 shows the configurable information types that we currently use with the approach. Note that this figure depicts part of the UML meta-model extracted to show the different types of supported model element and properties of such that we are interested in. Figure 3 depicts one of the development activities that we are able to recognize, the conversion of an attribute into a class. The figure shows four elementary changes and the change events generated from these. Within the figure, the events are simplified in terms of properties to show only a part of each event.

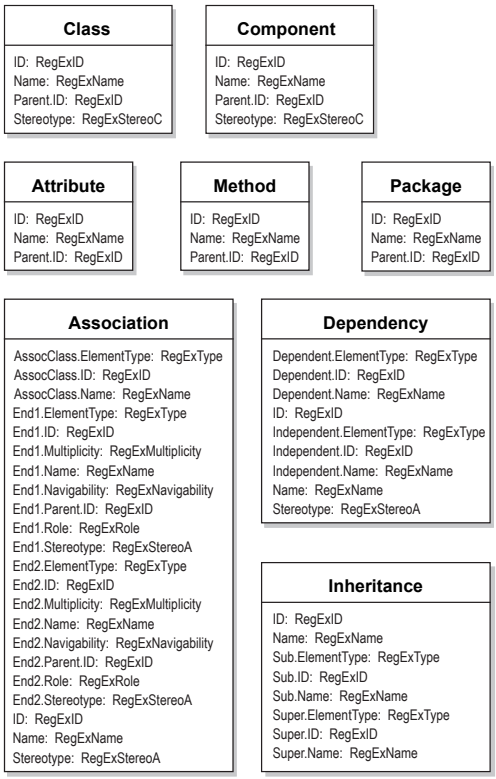


**Figure 2. Configurable information types to capture changes to structural UML models**

## 4 Recognizing development activities

The challenge in recognizing the development activities applied to a model based on underlying elementary changes is variability. Not only can different orders of the same elementary changes establish a single development activity, but different types and numbers of elementary changes can establish the same development activity. There are many permutations to consider. We are therefore not able to instantly recognize development activities as changes are applied to a model; it is necessary to predefine what we want to recognize and process incoming events within this context.

Our strategy for creating a rule catalog suitable for recognizing development activities was iterative, as illustrated in Figure 4. We derived our set of development activities based on a systematic study of the related literature and according to our own experience from performing and consulting on UML-based software development projects. We do not consider the current set of rules to be complete, but it is being improved with every use and appears to be stabilizing.

The current set comprises thirty-eight development activities (twenty-one rules with sixty-seven alternatives). Development activities that apply to relations include: refining an unspecified association into one or two directed associations; 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: splitting a class; converting

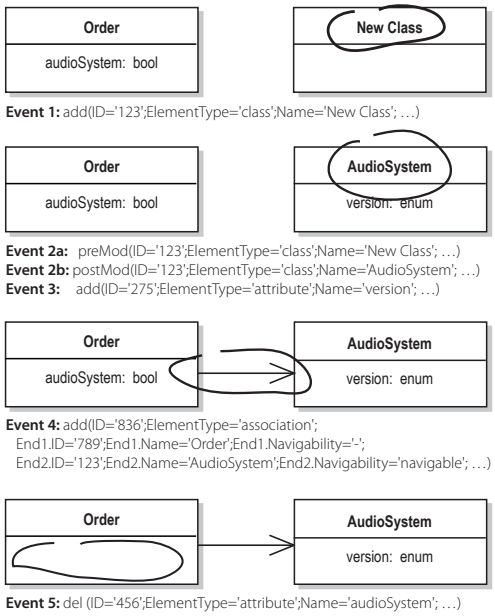


**Figure 3. A recurring development activity with simplified thrown change events**

an attribute to a class; refining an unspecified association into one or two directed associations; resolving one to many associations; and resolving association classes.

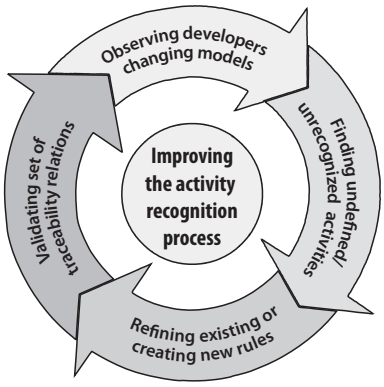


**Figure 4. Rule improvement process**

## 4.1 Rule definition

Since our approach depends upon each development activity to be recognized in terms of its constituent elementary changes, one aim was to create rules that allow for the necessary variability described above without the necessity to define exactly each possible permutation. To achieve this, we rely on the concept of masks.

**Masks** We define the elementary changes that comprise a recognizable development activity as masks. Each property that needs to be compared is defined within the mask. Values of the properties can be defined as static expected values or as references to the properties of another mask within the same development activity. The latter means that the value is expected to be the same as the value of the referenced property of a change event that has been already assigned to a different mask of the same development activity. These references between masks allow elementary changes to be related to one another.

A mask therefore defines those properties of a matching change event that have to take certain values and those that may take any value. By laying masks over an incoming event, one can determine whether the event matches a step of an activity or not. By this mechanism, we are able to require a certain type of change to an element (i.e., add, delete or modify) and a certain state of the element's properties before or after the change to ease recognition. Since modifications often happen incrementally, with a developer changing some properties before realizing that another modification is necessary, a mask defined to require the changed element to take a certain state after a modification will allow for a number of modification events to the element before the actual matching event is incoming.

**Sequence of changes** At least two masks have to be defined as a change sequence to define a development activity. This is because to identify an activity one need to compare at least two states, before and after the change. The order of the masks within the change sequence does not imply any required order of the incoming events. As most masks are dependent upon other masks by referencing their values, it might not be possible to compare a matching change event immediately after its arrival. To address that circumstance, we hold a number of past incoming events in an *EventCache* and can assign matching events as soon as the mask is comparable, which means that all references of the mask can be resolved (see Section 5.2). There is one mask within each change sequence that must have no references. This mask is called the *TriggerMask*. The change defined by the *TriggerMask* is modifying or deleting the original model element and is the action that actually starts the development activity. As it is possible to create the enhancing or replacing structure before changing the original element, the event matching the *TriggerMask* is not necessarily the first incoming event. The algorithm described in Section 5.2 starts to compare additional development activities with an incoming *TriggerEvent*.

**Alternative sequences** To address the issue that it is possible to perform a development activity in multiple ways, we group several change sequences as one rule able to recognize the same development activity.

### 4.2 Rule representation

We define our rules in the open XML format. A self-defined XML Schema Definition (XSD) gives the syntax of a rule. 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 Section 4.1). Each alternative is composed of a <ChangeSequence> and a <LinkUpdate>. The <ChangeSequence> consists of a definition of all expected elementary change events as masks to recognize the sequence. The <LinkUpdate> 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 5.3).

Masks are defined as <Event> sections within a <ChangeSequence>. Every event mask has a unique ID which is used to reference properties of other events during mask definition. There is the one special *TriggerEvent* ID="T" within each sequence that allows the rule engine to recognize the development activity. The remaining events are numbered starting with ID="1". An <Event> itself consists of property-value pairs for event type, element type and all the element properties that shall be compared with incoming events.

Listing 1 shows part of the rule to recognize the development activity of refining one unspecified association into two unidirectional ones. We use a special notation for the masks within the <Event> section of the listing to save space. Within the rule catalog, each of the comma separated property-value pairs is represented as a property tag with a value in XML notation. Alternative 2 of this rule assumes the developer to delete the existing association and to add two new ones. Alternative 3 assumes the modification of the existing association and the addition of one new one.

Since one development activity might be completely part of another (e.g., an unspecified association could be replaced by one directed association or by two such associations) we cannot define two distinct rules that permit us to recognize the one case from the other. It is necessary to have a rule for the partial activity to guarantee that an action is performed for it, but also a rule for the larger composite activity. It is only necessary during rule definition to keep in mind that the partial rule might already be fired and the interim action completed before the further rule is fired and the larger activity completed.

```
<Rule ID="2"> <!-- development activity: refinement of one
  unspecified association into two unidirectional ones -->
  <Alternative ID="1" ... />
  <Alternative ID="2">
    <ChangeSequence>
      <Event ID="T"> DEL('association', Navigability:'-' || 'bi')
        </Event>
      <Event ID="1"> ADD('association', End1.ID: T.End1.ID,
        End2.ID: T.End2.ID) </Event>
      <Event ID="2"> postMOD('association', ID:1.ID, End1.ID:
        T.End1.ID, End2.ID: T.End2.ID, Navigability:'uni') </Event>
      <Event ID="3"> ADD('association', End1.ID: T.End1.ID,
        End2.ID: T.End2.ID) </Event>
      <Event ID="4"> postMOD('association', ID:3.ID, End1.ID:
        T.End1.ID, End2.ID: T.End2.ID, Navigability:'uni') </Event>
    </ChangeSequence>
    <LinkUpdate>
      <UpdateSource> T.ID </UpdateSource>
      <UpdateTarget> 1.ID </UpdateTarget>
      <UpdateTarget> 3.ID </UpdateTarget>
    </LinkUpdate>
  </Alternative>
  <Alternative ID="3">
    <ChangeSequence>
      <Event ID="T"> preMOD('association', Navigability:'-' || 'bi')
        </Event>
      <Event ID="1"> postMOD('association', ID:T.ID, End1.ID:
        T.End1.ID, End2.ID: T.End2.ID, Navigability:'uni') </Event>
      <Event ID="2"> ADD('association', End1.ID: T.End1.ID,
        End2.ID: T.End2.ID) </Event>
      <Event ID="3"> postMOD('association', ID:2.ID, End1.ID:
        T.End1.ID, End2.ID: T.End2.ID, Navigability:'uni') </Event>
    </ChangeSequence>
    <LinkUpdate>
      <UpdateSource> T.ID </UpdateSource>
      <UpdateTarget> 2.ID </UpdateTarget>
    </LinkUpdate>
  </Alternative>
</Rule>
```

**Listing 1. Part of the rule to identify the refinement of one unspecified association into two unidirectional associations**

While defining rules, one also has to decide whether one wants to recognize an activity only if it preserves the initial semantics of the changed model element or if it is permissible to change the semantics during the activity. This will depend upon the rigor desired of the developer. For example, to preserve semantics one should not refine a directed association into two unidirectional ones. If we require development activities to preserve semantics, it is possible to be more certain in recognizing them, and the rules become more narrow and distinct. Where greater flexibility is demanded, the rules become more involved. By supporting both approaches for each rule, however, it is possible to inform the developer in the case where semantics are violated if desired, and this is the strategy we adopt.

### 4.3 Rule editor and validator

We developed a rule editor to assist with the tasks of rule creation, editing and validation. This has helped us to address common problems when evolving a rule catalog: (a) structural issues – by checking against a XML schema definition (XSD), inconsistencies within the structure of the rule catalog can be found; (b) element type and property inconsistencies – by checking against the current information model (see Figure 2), undefined properties of elements can be found; (c) syntax errors within property values – by applying regular expressions, the syntax of property values can be checked; (d) reference specification errors – as references relate to properties of other masks, it is necessary to check whether the event that will be assigned to the other mask will have that referenced property and that the referenced mask is specified within the change sequence; and (e) reference dependency errors – as the assignment of an event to a mask requires the resolution of all references within the mask, at least one mask without references (i.e., the *TriggerMask*) is necessary, along with an overall tree-like reference structure starting from this mask, with no cyclic dependencies.

## 5 Rule application

In Section 3, we described the process of generating a stream of change events reflecting elementary changes to model elements. In Section 4, we described the rules that hold patterns of change events that we want to find within a stream of events. The matching task requires an effective rule engine that is able to find matches between incoming change events and the predefined rules. Such reactive systems are built according to the Event-Condition-Action paradigm [12]. This paradigm simply defines systems that trigger an action after an incoming event has matched a defined condition. As we do not only want to react to a single event, but want to react to patterns over the event history, we need a much more sophisticated rule engine with complex event processing (CEP) [9]. The design of our rule engine needs to provide for the variability we want to permit (see Section 4).

### 5.1 Rule engine

The rule engine we developed consists of the following components (as illustrated in Figure 5). An *EventController* that receives new incoming change events, validates them against the current information model and handles all the following actions to process the

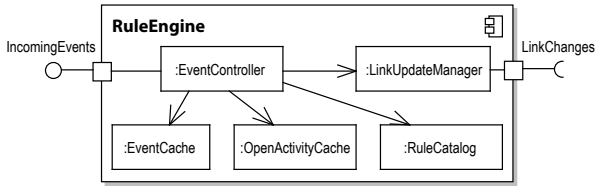


**Figure 5. Rule engine overview**

event. An *EventCache* that stores a configurable number of incoming events in a first in, first out buffer. An *OpenActivityCache* that stores all partly recognized, but still incomplete activities. The *RuleCatalog* that holds all the predefined rules. The *LinkUpdateManager* that generates link update commands according to a recognized development activity and invokes a request for user interaction or intervention if the update is ambiguous or problematic. How these components work together to handle incoming change events is described in the following section.

### 5.2 Handling an incoming event

The whole development activity recognition process is triggered by incoming change events. On the arrival of a new change event within the *EventController*, the event is validated against the current information model. This means that the type of the changed model element has to be defined within the information model and all defined properties of this element type have to be present within the event. Furthermore, the values of the properties are checked against regular expressions. After validation, the incoming event is passed to the *EventCache*, the *OpenActivityCache* and the *RuleCatalog*. The performed actions within each of these components are described in turn below.

**EventCache** The incoming change event is put in the *EventCache*. If full (the buffer size is configurable), the oldest event will be deleted, along with its occurrence within all *OpenActivities*. As all events are moving through the cache with each incoming event, every event has an age. Each combination of *preMod* and *postMod* events will be handled as one event within the buffer as they are triggered by the same elementary change. This means that both events have the same age and will be removed together from the buffer.

If the incoming change is a *DEL* event, all *ADD* and *MOD* events of the same model element will be removed from the *EventCache*. This minimizes the risk of recognizing false development activities. For

*DEL* events removed from the *EventCache*, still existing and hanging traceability relations pointing to the deleted element will be searched for and deleted. This means that the deletion is handled as a straightforward deletion of a model element. For an unassigned *ADD* event, at the time when it is going to be removed from the *EventCache*, the added element receives a configurable tag (e.g., new element) that can be used within the CASE tool to find and/or highlight those elements that might require the creation of new traceability relations. That mechanism reflects our assumption that events that cannot be assigned to a wider development activity are new elements within the model and need the creation of a trace, depending on what is specified in the project's traceability meta-model (if defined).

**OpenActivityCache** The incoming change event is assigned to all *OpenActivities* with a missing event equivalent to the incoming event. Each *OpenActivity* is an instantiation of one alternative of a rule and consists of a sequence of required change events defined as masks. These masks of *OpenActivities* can have one of three states: not assignable, assignable and assigned. The state of not assignable means that the mask has references to other masks that are not assigned with a matching event as yet (i.e., if we have references in the mask to the event that is going to be assigned to the referenced mask, we need to assign that event first). The state assignable means that the mask is comparable with incoming events and might be assigned once a matching event is incoming. The state assigned means that a matching event has been assigned.

The incoming change event is compared with all masks of all *OpenActivities* that are in the state assignable. On a match between the incoming event and one of the masks, the state of the mask will be changed to assigned and the mask will receive a reference to the matching event within the *EventCache*. Furthermore, all remaining masks of the rule with the state not assignable within the *OpenActivity* of the just assigned event are recursively tried to set the state to assignable. If one of these masks becomes assignable, the *EventCache* is searched for an event matching that mask. This mechanism is necessary to handle references to masks other than the TriggerMask.

If one or more *OpenActivities* could be completed by assigning change events, as described above, these activities will be deleted from the *OpenActivityCache* and passed over to the *LinkUpdateManager* to carry out the necessary traceability relation updates (see Section 5.3). The change events of the completed development activities will be kept within the *EventCache* to allow for the further recognition of partly identical activities, since one elementary change could contribute to a number of development activities concurrently.

**RuleCatalog** The *RuleCatalog* is searched for alternatives of rules with *TriggerEvents* matching the type and properties of the incoming change event. All matching alternatives are established as new *OpenActivities* and the *EventCache* is searched for matching events to complete the *OpenActivities*. It is configurable to instantiate only *OpenActivities* for *TriggerEvents* caused by changes to linked model elements to reduce the number of activities, but this might miss activities in those situations where the developer is creating traceability relations on the changed model element after performing a change.

**EventStore** On closing the model, the rule engine offers two ways to handle unmatched change events and *OpenActivities* within the cache. The user can choose to discard all events and to delete all hanging traceability relations associated with already deleted, but not fully replaced, model elements. It is also possible to store the current events within the *EventCache* and to restore these and all *OpenActivities* on the next start-up of the model. In that case, all the hanging traceability relations will be retained.

### 5.3 Triggering a predefined action

In Section 5.2, we discussed the handling of incoming change events within the rule engine and their assembly to recognize development activities. During that process, one or more *OpenActivities* might have been completed with the incoming event. Completed *OpenActivities* are passed to the *LinkUpdateManager*.

The *LinkUpdateManger* acquires information about the traceability relations on all update sources and all update targets defined within the rule. Depending on whether there are traceability relations on at least one of the update sources, a list of all involved (existent, inconsistent and not-yet-existent) traceability relations is created by the *LinkUpdateManger*. Each relation receives a tag with the necessary action to update it. In case there are relations with no distinct action to update, a user dialog displays the situation and impacted elements, and requires a decision for these cases. Detailed information about how the update is performed is outside the scope of this paper and will be presented in a subsequent paper. In addition to the maintenance of traceability, we can foresee support for other development tasks, one option being to use the information about the recognized development activity for the documentation of changes in a version control system.

## 6 Initial validation

The objectives of our validation were to: (OBJ 1) assess whether our rules account for all changes and the recognition of common development activities; and (OBJ 2) determine whether the correct rules are fired when there is variation in task execution.

### 6.1 traceMaintainer prototype

In order 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 that it imports from an XML file; (b) based on a match between events and rules, it restores traceability; and (c) the editing of existing rules and the specification of new rules.

The prototype has been designed to be independent of specific CASE tools. The intention is for it to be deployable with every CASE tool that allows for the capturing of the necessary change events to model elements and that allows for 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. We have developed adapters to ARTiSAN Studio and to Sparx Enterprise Architect, as well as a rule catalog for changes to structural UML models developed within these tools. The main purpose of the adapters is the generation of change events and the collection of element properties to provide the rule engine with standardized elementary change events.

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, such as EXTESSY ToolNet [5], 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, due to its ability to link every element of a model.

### 6.2 Case study

We performed the validation in two stages using the prototype and pre-existing analysis and design models of two systems created during an earlier experiment (see [10]). The first analysis model 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 to create and handle the models. The analysis models for both systems were given to two developers. Developer A had four years of industrial experience in model-based, object-oriented software development and developer B had two years of experience. Both had university-level education on the topic. The task was to refine the analysis model of the system into a design model that could be implemented. We used these two initial analysis and four resulting design models for our current study. Table 1 provides some statistics on the elements of the models.

**Table 1. Number of elements within models**

| | Library system | | | Wiper system | | |
|---|---|---|---|---|---|---|
| | Anal | Design | | Anal | Design | |
| | ysis | A | B | ysis | A | B |
| Classes | 23 | 35 | 33 | 21 | 31 | 34 |
| Attributes | 64 | 67 | 64 | 35 | 36 | 35 |
| Methods | 50 | 50 | 48 | 28 | 32 | 29 |
| Associations | 20 | 39 | 40 | 18 | 36 | 39 |
| Generalizat. | 10 | 4 | 4 | 5 | 3 | 3 |
| Elements | 167 | 195 | 189 | 107 | 138 | 140 |

To evaluate our approach to development activity recognition, we manually defined two scenarios (change execution paths) to transform between each analysis-design model pair (giving eight scenarios in all). To explore the greatest possible variation in the changes that would allow for the transformation, we created two scenario types. The first reflected those change activities that would have the least impact on the model's structure (i.e., if it was possible to modify an element, instead of deleting and recreating it to reach the same structure, then that was the strategy adopted). Conversely, the second reflected those changes that would have the most impact on the structure. Table 2 shows the number of elementary changes and the number of development activities comprising each scenario.

**Table 2. Changes and development activities**

| | Library system | | | | Wiper system | | | |
|---|---|---|---|---|---|---|---|---|
| | A | | B | | A | | B | |
| | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 |
| *ADD* | 45 | 84 | 34 | 62 | 55 | 98 | 52 | 87 |
| *DEL* | 17 | 56 | 12 | 40 | 24 | 67 | 19 | 54 |
| *MOD* | 74 | 52 | 55 | 38 | 47 | 32 | 51 | 34 |
| Total | 136 | 192 | 101 | 140 | 126 | 197 | 122 | 175 |
| Activ. | 35 | 49 | 24 | 37 | 32 | 46 | 28 | 41 |

Given the eight scenarios, and knowing the development activities intended by the elementary changes, we applied them to our rule engine and catalog. This was

simplified since the rule engine offers a parser that can read scenarios of formerly captured change events from a text file instead of having to perform all the changes interactively within a CASE tool. This was developed to help us improve the rules (as per Figure 4). For this study, we used a buffer size of thirty events for the incoming *EventCache*, as this had proved adequate in previous trials. The objective was to see whether our tool and approach identified all the change events and intended development activities accurately (OBJ 1).

OBJ 2 was to examine the ability of our approach to recognize development activities when developers do not work as expected. We used the same eight scenarios but applied the elementary changes five times in random orders, with a few exceptions where structures were built upon others and required a defined order. To apply the change events in random order means that the extraction of an attribute may have been started at the beginning of the scenario and not be completed until the end, so we used a buffer size of two-hundred events, large enough to store the whole scenario.

### 6.3 Results and discussion

Since our rule catalog was evolving, we used two of the scenarios to pre-examine the representativeness of the rules for the study. In so doing, we made minor adjustments to two of the rules to improve the catalog. Examining the results achieved when assessing the remaining six scenarios, we found that the prototype was able to recognize, with total accuracy, all of the changes and intended development activities. Further, the five different orders in which we conducted each of the eight scenarios made no difference, as we were still able to recognize all the expected development activities. We can say that our rules catalog was sufficient for the context and captured the variety of ways in which the analysis model could have been transformed by a practitioner into the final design model. While we did not encounter any performance problems, an exception could be envisaged when a change triggers a large number of events (e.g., deleting a package with hundreds of elements), but in trying such a setting we found little appreciable delay. The relation between the size of the model and the performance of the approach, and further model samples, is the subject of ongoing studies.

The computational effort required to support our approach depends more upon: (a) the number of defined rules and on how many new development activities can be triggered at the same time with an incoming change event; (b) the size of the buffer, as this determines the time interval over which the attempt to tie incomplete activities with incoming change events will be made; and (c) how long it takes to perform the desired action when a development activity is recognized. To address (a), the number of rules is limited and the trigger can be defined to differentiate as much as possible. For (b), we assume that most developers will not be able to handle more than a few development activities in parallel, so a size of thirty to fifty events should suffice. An efficient update solution addresses (c). All these topics are subject to ongoing investigation.

The results from this case study are preliminary and there are threats to validity. Given a small set of models, it is possible that the changes to these models are not representative of a wider population. Also, it might be possible to execute the changes between both states of the models in such a way that not all activities would have been recognized.

## 7 Related work

There is related work on categorizing and identifying changes to UML models. Engels et al. present a classification of UML model refinements [4] to preserve consistency during the evolution of UML-RT models (a UML enhancement for real-time systems). The authors identify three kinds of modification: creation, deletion and update and the focus is limited to four model elements: capsules, ports, connectors and protocols. The work does not show how these atomic changes can be combined into the recognition of composite change activities with development intention. Furthermore, the approach does not appear to deal with dynamic (i.e., run-time) evolution of models as we do.

Shen et al. [11] suggest an extension to the UML meta-model via specified stereotypes according to four types of refinement. Using these stereotypes on different abstraction levels of a project, they are able to check consistency between levels. The approach expects the stereotypes to be set manually by the developer. By using our approach, and maintaining traceability between abstraction layers, this is not necessary.

Hnatkowska et al. [7] specify behavioral refinements in UML collaboration diagrams and describe how these relate to structural refinements. The purpose is to establish refinement relationships between different abstraction layers. The authors provide a classification of nine simple class diagram refinements (e.g., adding a class), so similar to elementary changes and development activities. However, the authors do not describe how these refinements could be detected and require the developer to establish them manually at present.

Cleland-Huang et al. [2] present, as one aspect of their work, the maintaining of traceability between requirements after predefined changes to requirements.

There are similarities between their 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, and they identify seven possible change activities to requirements. All these activities consist of only one elementary change as is suitable for requirements management tools with high-level change functions. For changes to complex models created using UML, the recognition of change becomes more difficult, as more specifically addressed within this paper. In [10], we focus on the problem of maintaining a set of traceability relations in the context of evolutionary change and limit this to post-requirements relations in UML-driven development. That paper presents a survey of related traceability work. Within the current paper, we focus primarily on the recognition of development activities applied to UML models, narrowing the scope to give us the opportunity to discuss and describe the recognition part in the necessary depth.

## 8 Conclusions and future work

This paper presents an approach, supported by a prototype tool (traceMaintainer), to tackle the recognition of development activities applied to models in the context of UML-based software development. Activity recognition rules have been defined to cope with the high variability in task execution and a rule engine has been developed to apply these rules to a flow of elementary change events captured from within a CASE tool. The main motivation for identification of these development activities is to maintain the viability of existing traceability as a project proceeds, so as to address the problem of traceability decay.

Though we made a conscious effort to be as exhaustive as possible in terms of the rules we use to identify possible development activities undertaken with UML models, the approach and rule catalog will be refined as we gain more experience, especially by applying our rules to additional industrial case studies, thereby gaining more empirical data on performance and related factors. Our approach and tool have been validated initially through a small case study, focusing specifically on the task of automatically recognizing those development activities that have implications for traceability, thus providing some demonstration of feasibility and practicality. The early results are encouraging as it is shown that, with the current rules, it was possible to recognize all the development activities applied to the models in our study, and we gained some evidence that the rule catalog is converging. Since the definition of rules can be very challenging, and needs an understanding of their syntax and semantics, we are currently investigating how to perform an activity in several different ways within the CASE tool and to use that data to automatically create a rule (i.e., an automated rule recorder) to promote wider applicability. We are also investigating the possibility of applying rules in a forward and backward direction to handle the undo function within the CASE tool whilst still recognizing development activities.

**Acknowledgments** This work is partly funded by Deutsche Forschungsgemeinschaft (DFG) id Ph49/7-1.

## References

[1] P. Arkley and S. Riddle. Overcoming the traceability benefit problem. In *Proc. 13th Int'l Requirements Eng. Conf.*, pages 385–389. IEEE Computer Society, 2005.

[2] 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 CS, 2002.

[3] A. Egyed, P. Grünbacher, M. Heindl, and S. Biffl. Value-based requirements traceability: Lessons learned. In *Proc. 15th Int'l Requirements Eng. Conf.*, pages 115–118, 2007.

[4] G. Engels, R. Heckel, J. M. Küster, and L. Groenewegen. Consistency-preserving model evolution through transformations. In *5th Int'l UML Conf.*, volume 2460 of *LNCS*, pages 212–226, Dresden, 2002. Springer.

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

[6] 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.

[7] B. Hnatkowska, Z. Huzar, L. Kuzniarz, and L. Tuzinkiewicz. Refinement relationship between collaborations. In *Proc. WRKUMLCP II*, pages 51–57. IEEE CS, San Francisco, USA, 2003.

[8] K. Lano. *Advanced systems design with Java, UML, and MDA*. Elsevier, Amsterdam, Netherlands, 2005.

[9] D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Professional, Reading/MA, May 2002.

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

[11] W. Shen, Y. Lu, and W. L. Low. Extending the UML metamodel to support software refinement. In *Proc. WRKUMLCP II*, pages 35–42. IEEE CS, San Francisco, USA, 2003.

[12] J. van Bemmel, P. Dockhorn, and I. Widya. Paradigm: Event-driven Computing. White paper TI/RS/2004/051, Lucent Technologies, CTIT, 2004. `https://doc.telin.nl/dscgi/ds.py/Get/File-48190`.