traceMAINTAINER – Tool Demonstration

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. This paper outlines a proposed demonstration that will present the core functionality of *traceMAINTAINER*, a prototype tool that enables the semi-automated maintenance of traceability relations held between different models of software systems expressed in UML. The demonstration will cover: how *traceMAINTAINER* integrates with UML modeling tools, such as Enterprise Architect; how traceability updates are handled based upon the recognition of development activities; what happens when decisions about a traceability update cannot be made fully automatically; and how to tailor the rules that guide the traceability maintenance process.

1 Introduction

traceMAINTAINER is a prototype tool that enables the semi-automated maintenance of traceability relations held between different models of software systems expressed in UML ([1], [2]). The approach analyses changes undertaken to structural UML models while working within a CASE tool that supports UML modeling. Change events are captured and sequences of events are sought that correspond to predefined rules. These rules represent the various ways in which recurring development activities can be undertaken and directives to update the impacted traceability relations once identified. This paper is provided as a companion to the architectural description of *traceMAINTAINER* provided in [3].

2 Integration with Case Tools

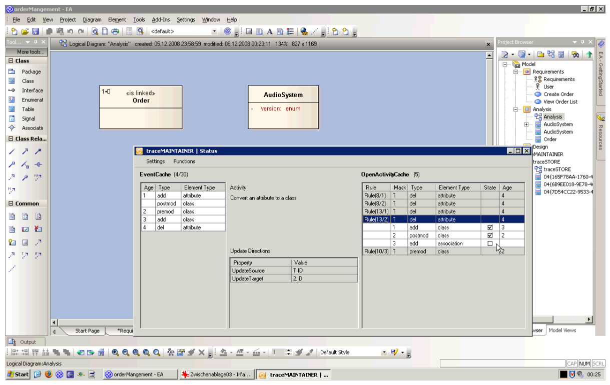
traceMAINTAINER is designed to work with traditional UML modeling tools to expand their support for traceability. The demonstration will begin by illustrating *traceMAINTAINER*'s integration with Enterprise Architect and by explaining how such integration can be extended to other UML modeling tools. The integration relies upon two specific components: an event generator that captures changes to supported model elements and a *traceSTORE* that extends the traceability functionality of the base CASE tool. *traceSTORE* holds traceability relations in an additional class model within the development model and we will discuss the benefits of this realization over the vanilla traceability functionality (of Enterprise Architect in this case). We will then create a small UML model for demonstration purposes. This will involve creating a use case model with two use cases, relating these to elements of a design model, and demonstrating *traceSTORE*'s ability to show and navigate the traceability relations in both models. We will also emphasize the role of the indicators that we have introduced to recognize the existing traceability relations on an element and their count. We will further explain the structure of the element references and relations while creating the example model.

3 Recognizing Development Activities that Impact Traceability

In this part of the demonstration, we will show how *traceMAINTAINER* is used in the context of software development. Expanding the above example, we will show how an analysis model may need to be changed based upon a change request relating to a use case. We will use the development activity of extracting an attribute into its own class to discuss the functionality resulting from this change request and the use of *traceMAINTAINER* to account for it. We will then provide a step-by-step walkthrough as to the tasks undertaken and the response of *traceMAINTAINER* to automatically maintain the traceability. During these activities, we will show the rule engine status window (see Figure 1) to explain how the development activities are recognized by *traceMAINTAINER*. We will also explain how the traceability relations are updated after recognizing the development activity and the kind of feedback the user receives.

4 Semi-automated Traceability Maintenance

In the preceding parts of the demonstration, we will have presented the use of *traceMAINTAINER* in a context were no user interaction was necessary (automated traceability maintenance). Nevertheless, while development activities can be recognized by *traceMAINTAINER*, it is not always possible to make a definitive update of traceability relations. In the next part of the demonstration, we will present an example according to Figure 2 in which user interaction is required. After recognizing the development activity, *traceMAINTAINER* presents the user with a dialog that lists all the existing and potentially new traceability relations involved in the activity. The user is required to decide how to handle certain relations. We will explain when there is the necessity for user interaction, as well as the reason for the different alternatives presented. After choosing the preferred alternative, we will show how the traceability relations then get updated (as per the previous section).



EventCache (4/30)

Age	Type	Element Type
1	add	attribute
	postmod	class
2	premod	class
3	add	class
4	del	attribute

Activity

Convert an attribute to a class

Update Directions

Property	Value
UpdateSource	T.ID
UpdateTarget	2.ID

OpenActivityCache (5)

Rule	Mask	Type	Element Type	State	Age
Rule(8/1)	T	del	attribute		4
Rule(8/2)	T	del	attribute		4
Rule(13/1)	T	del	attribute		4
Rule(13/2)	T	del	attribute		4
	1	add	class	☑	3
	2	postmod	class	☑	2
	3	add	association	☐	
Rule(10/3)	T	premod	class		2

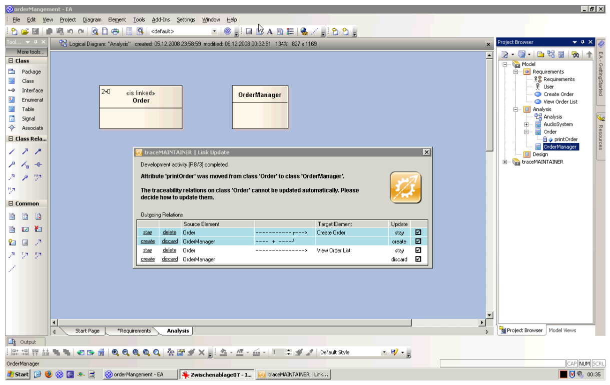
Fig. 1. Before performing the last change of the development activity, the *traceMAINTAINER* status window shows that the correct rule has been instantiated as an OpenActivity and that all already performed changes have been correctly assigned to it. The remaining mask is comparable and requires creating an association between both classes involved in the development activity.

5 Changing Traceability Maintenance Rules

The current rule catalog used by *traceMAINTAINER* has been in use for about one year in different experiments and by industry partners. During this period it has been stable. However, it is unlikely that this set of rules is fully complete and correct. Furthermore, there are development activities that may be carried out in different ways depending on the domain of the project and the pre-disposition of the developers. It is therefore desirable to be able to customize the existing rules and to support the traceability of new model elements. In this part of the demonstration, we will present the underlying rule catalog of *traceMAINTAINER*. We will describe how these rules were derived and function. We will also illustrate how they are stored and describe their internal structure. We will further create a new rule and, using this example, discuss the concept of properties, masks, alternatives and traceability updates (all key concepts underlying the traceability maintenance process supported by *traceMAINTAINER*). We will demonstrate different ways to define properties and explain how they help in creating *good* rules. Finally, we will highlight the process for validating new or changed rules.

6 Status

traceMAINTAINER provides an extensive set of features for maintaining traceability between a broad spectrum of UML model element types. Results show that the approach is capable of reducing the effort (and so the cost) of maintaining traceability quite dramatically and at quality levels comparable to manual



traceMAINTAINER | Link Update

Development activity [R8/3] completed.

Attribute 'printOrder' was moved from class 'Order' to class 'OrderManager'.

The traceability relations on class 'Order' cannot be updated automatically. Please decide how to update them.

Outgoing Relations

		Source Element		Target Element	Update	
stay	delete	Order	----------┌--->	Create Order	stay	☑
create	discard	OrderManager	---- + ----┘		create	☑
stay	delete	Order	--------------->	View Order List	stay	☑
create	discard	OrderManager			discard	☑

Fig. 2. *traceMAINTAINER* dialog that informs the user about a recognized development activity and requires a decision to be made between several options in those cases where the traceability maintenance cannot be determined automatically.

maintenance. The approach is intended as a complement to those approaches that initially create traceability relations using either manual or automated techniques. The reader is directed to a companion publication for a detailed architectural description on *traceMAINTAINER* [3].

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

References

1. 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)
2. 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)
3. Mäder, P., Gotel, O., Philippow, I.: traceMaintainer: A Tool for the Semi-automated Maintenance of Model Traceability. In: (submitted), Enschede, Netherlands (June 2009)