Published in Proceedings 16th International Conference on Model-Driven Engineering Languages and Systems (MODELS 2013), Miami, FL, USA.
@INPROCEEDINGS{conf/models/kuschkeMR13, title={Recommending auto-completions for software modeling activities}, author={Kuschke, Tobias and M{\"a}der, Patrick and Rempel, Patrick},booktitle={Model-Driven Engineering Languages and Systems}, pages={170--186},year={2013},publisher={Springer},doi= {10.1007/978-3-642-41533-3_11}}

Kuschke, Tobias, Patrick Mäder, and Patrick Rempel. "Recommending auto-completions for software modeling activities." Model-Driven Engineering Languages and Systems. Springer Berlin Heidelberg, 2013. 170-186.

Recommending Auto-Completions for Software Modeling Activities

Tobias Kuschke, Patrick Mäder and Patrick Rempel

Department of Software Systems, Ilmenau Technical University
{tobias.kuschke|patrick.maeder|patrick.rempel}@tu-ilmenau.de

Abstract. Auto-completion of textual inputs benefits software developers using IDEs and editors. However, graphical modeling tools used to design software do not provide this functionality. The challenges of recommending auto-completions for graphical modeling activities are largely unexplored. Recommending auto-completions during modeling requires detecting meaningful partly completed activities, tolerating variance in user actions, and determining the most relevant activity that a user wants to perform. This paper proposes an approach that works in the background while a developer is creating or evolving a model and handles all these challenges. Editing operations are analyzed and matched to a predefined but extensible catalog of common modeling activities for structural UML models. In this paper we solely focus on determining recommendations rather than automatically completing an activity. We demonstrated the quality of recommendations generated by our approach in a controlled experiment with 16 students evolving models. We recommended 88% of the activities that a user wanted to perform within a short list of ten recommendations.

1 Introduction

Automating tasks of a software engineering process is a state-of-the-art way to increase the quality of a software product and the efficiency of its development. Auto-completion of textual inputs, as it exists in source code editors of modern integrated development environments, supports developer's work without the need to interrupt code writing and triggering menu functions, making its usage very efficient.

However, when designing a system in a graphical modeling environment no such support is currently available. The challenges that arise when recommending auto-completions for such modeling activities are largely unexplored. Though, there are plenty of opportunities for supporting recurring activities during model-driven architecture and design. For example, Arlow and Neustadt [1] describe typical activities that have to be carried out when refining an initial UML analysis model into a design model for a system. Furthermore, many of Fowler's [2, 3] well-known source code refactorings impact the structure of a system and can as well be executed on a class model perspective of a system.

Recommending relevant auto-completions during graphical modeling requires handling challenging aspects accompanied with the problem such as:

C1 *Detect Partly Performed Activities.* Complex modeling activities are described by a set of editing operations with mutual dependencies. Detecting partly performed activities requires to match arbitrary incomplete subsets of editing operations while tolerating dependencies to unavailable information.
C2 *Tolerate Modeling Variances.* Modeling activities need to be detected in a variety of combinations of editing operations establishing the same activity. Not only can different orders of the same operations establish an equal activity, but different types and counts of operations can also establish the same activity.
C3 *Be Unintrusive.* A successful approach requires processing without noticeable system response delays.
C4 *Recommend Valid and Relevant Activity Completions.* Detected partial activities are not necessarily completable, i.e., not all are valid as recommendations. Furthermore, high numbers of valid recommendations have to be reduced to a limited set of most relevant completions for being useful.
C5 *Be Extensible for New Activities and Platforms.* A successful approach needs to be extensible to new activities and adaptable to other development tools.

We present an approach that handles these five challenges. The focus of this paper is determining and ranking relevant recommendations. In a follow-up publication we will focus on the auto-completion of a recommendation accepted by a user. Our paper is organized as follows. Section 2 reviews relevant related work on recognizing modeling activities, recommending modeling activities, and on auto-completion of modeling activities. In Section 3 we introduce our catalog of common modeling activities for structural UML models. Our approach for computing relevant recommendations of activity completions is introduced and discussed in Section 4. In Section 5 we evaluate the approach and assess its capabilities, followed by Section 6 where we discuss the results. Finally, Section 7 concludes our work and outlines future research.

2 Related Work

Our approach consists of three main stages: i) recognizing partial modeling activities, ii) recommending modeling activities, and iii) auto-completing modeling activities. In the following, we discuss previous research in these three areas.

Recognizing Modeling Activities. Sun et al. [4, 5] suggest model transformations based on pattern matching. A repository holds model transformation patterns, which can be extended through live-demonstrations of the user. Developers can select these patterns when modeling. The system then calculates and presents all automatically executable transformations where pattern preconditions match the current state of model objects. This approach provides a simple and comfortable way to define transformation patters and to share them with others. However, detecting partly executed transformations for automatic completion is not supported. Furthermore, the performance and usability of the approach is limited due to very high numbers of occurring transformation pattern matches.

Filtering and ranking matched transformations is not considered. The Eclipse framework *VIATRA2* by Rath et al. [6] presents another approach for live model transformation. *VIATRA2* can incrementally synchronize a target model to editing operations carried out on a source model. The authors employ an efficient RETE-based pattern matching technique [7]. Their transformation language supports incremental transformation rules as well as complex graph transformations. While this language could be used to express modeling activities, *VIATRA2* is not designed to detect partly performed states of activities. However, our approach uses similar concepts for the recognition of partly performed activities.

Recommending Modeling Activities. Several authors developed approaches to assist users of development tools with recommendations. Murphy-Hill et al. [8] recommend Eclipse commands. The applied data mining algorithms are efficient for recommending single commands, but are unsuitable for recognizing multi-step modeling activities. Recommendation ranking is based on user history, which could also be beneficial for modeling activities once long-term context information is available. *Strathcona* by Holmes et al. [9] recommends source code examples for using APIs. The user selects a source code fragment within Eclipse and starts the tool. Structural and context facts are extracted from the code and sent to a server. Based on four predefined heuristics the server matches the queried fact set to the stored examples trying to find structurally similar source code. The 10 most relevant examples are returned and presented.

Auto-Completion of Modeling Activities. Forster et al. [10] proposed *WitchDoctor* for detecting and completing source code refactoring's while observing developers writing source code. Their approach matches editing operations of the code to a list of refactoring's with every keystroke of a developer. Upon a match, the complete refactoring is being calculated and displayed as gray-colored suggestion within the editor. Similar to our approach, the authors capture atomic editing operations and match them against predefined patterns of operation sequences. However, the authors do not discuss how to extract valid and relevant recommendations within a set of detected activities. This becomes crucial when recommending a number of different modeling activities consisting of similar editing operations. Mazanaek et al. [11, 12] studied auto-completions for model and diagram editors in general. Based on graph grammars, their approach calculates all possible completions for incomplete model graphs. Although, the approach recommends correct structural completions for the current graph state, it does not support the completion of complex modeling activities within structural UML models. As such activities contain specific conditions regarding structural aspects and object property values it would be a difficult task to express them by a general graph grammar. Furthermore, it is impracticable to calculate and present all completions for a complex structural UML model. Sen et al. [13] propose a similar approach. A domain-specific modeling language and a partial instance of an appropriate model are transferred into an Alloy constraint model. This Alloy model is taken as input for a SAT solver that generates possible completed models. The system is triggered by the user and presents recom-

mendations in graphical form. It is mainly designed to support small modeling languages and computes its results within seconds up to minutes. The approach shows similar limitations as the previous. Furthermore, there are approaches [14, 15] that deal with user assistance in keeping models consistent and well-formed. Similarly, these approaches also calculate editing operations that are presented to the user. In contrast to our work, the focus of those works is changing a model in a minimal way in order to fix local inconsistencies rather than predicting a user's intent in performing complex modeling activities.

Summarizing, prior approaches that focused on recommending or on auto-completing activities within textual or graphical development environments all show limitations concerning the challenges identified in Section 1. In this paper, we present a novel approach for recommending valid and relevant completions of structural UML modeling activities while they are performed by a developer. Our contributions comprise the following aspects: 1) detecting partly performed complex modeling activities while observing developers editing a model, 2) tolerating variable editing operation combinations that establish the same activity, 3) recommending relevant modeling activity completions with every model editing operation that is carried out by a developer, 4) filtering and ranking the most relevant activity completions from a large number of possible recommendations, 5) processing without noticeable system response delays, 6) being extensible for defining new complex activities, and 7) being platform-independent to support different editors especially industrial modeling tools like Sparx Enterprise Architect [16] and IBM Rational Software Architect [17].

3 Modeling Activities and Illustrating Example

The basis for our approach is a catalog of predefined modeling activities for structural UML models that we created for our *traceMaintainer* approach [18, 19]. *traceMaintainer* recognizes meaningful modeling activities within incremental editing operations to traced UML models and semi-automatically updates impacted traceability relations. The activity catalog has been used and improved during several studies and experiments. Activities in the catalog are declared as patterns $AP = (ap_1, \ldots, ap_n)$ that describe a set of expected editing operations $EO = (eo_1, .., eo_i)$ each. The catalog comprises a set of 19 activity patterns with 67 alternative editing operation sequences to carry them out. These patterns cover 38 modeling activities. Examples of defined activity patterns are:

- Replacing an association between two classes by an interface realization (ap_6)
- Extracting an attribute into an associated class (ap_{13})
- Specializing an element inheriting to an sub element (ap_{17})

We introduce a simple modeling example and use it throughout the paper. A developer wants to enhance a small embedded system containing a *Control Unit* and a *Communication Adapter* for sending and receiving messages (see Figure 1, left). As part of the enhancement, two types of communication protocol shall be supported by the *Communication Adapter*: universal asynchronous

receiver/transmitter (UART) and serial peripheral interface (SPI). Furthermore, an exchanged message shall identify its sender and receiver allowing adding additional communicating units. A possible realization is shown in Figure 1 (right).

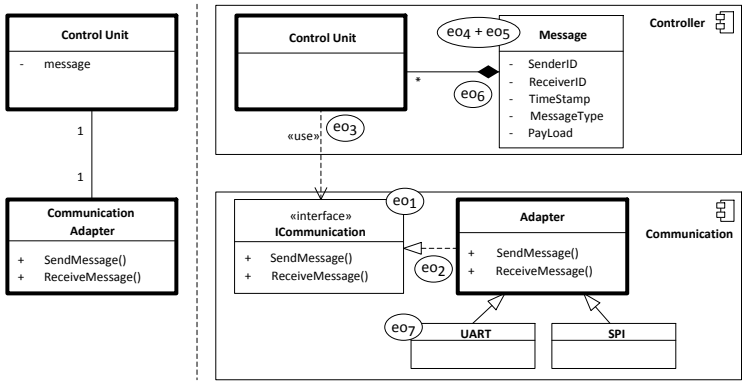

Fig. 1. Model of the illustrating example. On the left hand side the initial model state is show, while the right hand side depicts a possible enhancement.

As first step, the developer converts the association between *Communication Adapter* and *Control Unit* into an interface. Figure 2 shows from left to right the temporal progress of performed editing operations to implement the desired interface. First, she adds a new interface *ICommunication* (eo_1). One minute later, she adds a realization dependency between *Communication Adapter* and the new interface *ICommunication* (eo_2). Another minute later, she adds a use-dependency between *Control Unit* and the new interface *ICommunication* (eo_3).

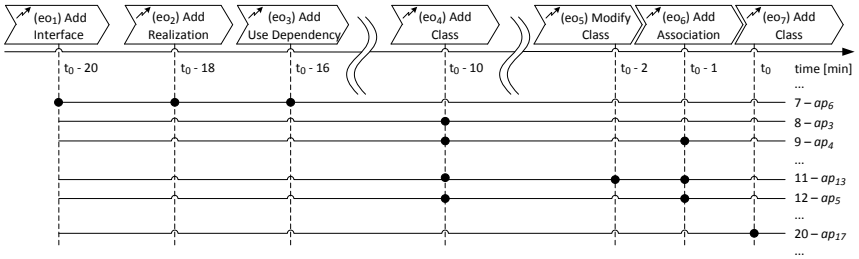

Fig. 2. Visualization of incoming events for the editing operations of the illustrating example, there temporal order, and a subset of matched activity patterns.

After a five minute break for planing the next step, the developer decides to extract the attribute message from the *Control Unit* into a separate class in order to extend it with additional properties. She starts this activity by adding a new class tot the model (eo_4). At this time she gets interrupted by a phone

call that takes 15 minutes. She continues by modifying the new class (eo_5) and by associating it to the *Control Unit* class (eo_6). Eventually, she creates a new class *UART* (eo_7). Our example stops at this point, which we will refer to as t_0 in the remaining text.

4 Approach

In this paper, we propose an approach for recommending valid and relevant completions of modeling activities for structural UML models while a developer is changing a model. In order to address the challenges identified in Section 1, we propose the following four step process. Each step is discussed in detail in the following four subsections.

Step 1: *Recognizing partly performed modeling activities.* While a developer is modeling within a tool, each editing operation is triggering an event containing detailed information of the change. Incoming events are matched against predefined activity patterns AP in order to detect partly performed modeling activities. The resulting set of activity candidates $AC = (ac_1, \ldots, ac_n)$ serves as input to the following process step.

Step 2: *Filtering invalid activity candidates.* All activity candidates that cannot be completed within the user's model are treated as invalid and filtered from the set AC.

Step 3: *Ranking activity candidates by relevance.* The filtered set AC is then ranked in relation to the relevance of each candidate for the user. Relevant are activities that the user wants to perform. Three ranking criteria are used to estimate relevance.

Step 4: *Presenting recommendations.* Finally, the filtered and ranked activity candidates in AC are reduced to a comprehensible number of recommendations and presented within the modeling environment.

4.1 Step 1: Recognizing Partly Performed Modeling Activities

We previously developed a traceability maintenance approach called *traceMaintainer* [19]. By recognizing modeling activities within incremental editing operations to traced UML models, *traceMaintainer* performs required updates to impacted traceability relations semi-automatically. While the goal of recognizing activities is similar to the approach presented here, there are fundamental differences in terms of required event processing. The need for matching partly performed activities required us to adopt a more advanced event processing. While our previous activity patterns contained designated trigger operations that had to occur in order to start the matching process, we required for this approach a mechanism that could match activity patterns starting from the first incoming event that contributed to them. Within the following paragraphs we introduce the redesigned recognition process.

First, each editing operation triggers an event of type *add*, *delete*, or *modify*, which carries the properties of the edited model element (see Figure 3, left).

Second, events are matched against a set of predefined activity patterns AP. Each activity pattern ap_x defines a set of expected editing operations $EO = (eo_1, .., eo_i)$ that have to be performed in order to complete a modeling activity. The definition of an expected editing operation comprises conditions that have to be fulfilled by an incoming event to be matched. For example, the event for editing operation eo_5 in Figure 3 (left) would be matched by the definition $E3$ in the activity pattern (right), if all conditions can be evaluated to true. Thus, the completion of an activity would be recognized if a sequence of incoming events matches all editing operations EO of an activity pattern.

We decided to implement our matching process on a RETE-based rule engine [7]. This is a well-known technique for the kind of problem we had to solve. RETE translates and merges all complex pattern descriptions into a network of condition checking nodes. The technique reaches high execution performance for large numbers of received events, because checking results are temporarily cached in the network. Paschke et al. [20] published a survey on rule-based event processing systems and identified the freely available RETE-based Drools platform [21] as being very efficient for complex event processing (CEP). Drools's implementation is mature and the rule declaration language is highly expressive. Furthermore, the platform allows to retract inserted events and to output all matched subsets of events.

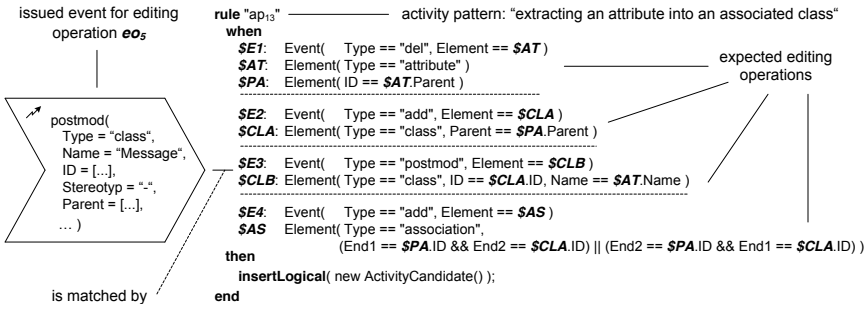

Fig. 3. Event for the editing operation eo_5 of the illustrating example (left) and the Drools rule declaration for activity pattern ap_{13} (right).

To integrate Drools in our solution, the activity patterns of AP with all their possible alternatives had to be declared using Drools's rule language. Figure 3 (right) shows a simplified Drools rule matching the ap_{13} activity pattern. The definition of an expected editing operation eo_x is separated into conditions for a matching event and for the edited model element. Cross-references between element properties are highlighted in bold. In order to recognize partly performed activities, all possible permutations of the expected editing operations in EO are declared within separate rules. Each fully matched Drools rule generates an activity candidate ac_x in AC, which defines the remaining editing operations for completing the modeling activity. As different modeling activities can contain

similar definitions of editing operation it is not possible to declare these activity patterns without partly overlapping each other, i.e., incoming events can be matched to multiple patterns in AP. Furthermore, an activity pattern can be recognized multiple times, because the RETE algorithm matches all possible event combinations that fulfill the pattern's conditions. Accordingly, the raw output AC of the activity recognition step requires post-processing steps to generate relevant recommendations for a user. In the second column of Table 1 we show the raw output AC produced for our illustrating example. A total number of 21 activity candidates has been matched based on the last triggered event at t_0 and all previously incoming events.

4.2 Step 2: Filtering Invalid Activity Candidates

Recommended activity candidates need to be completable. We call a candidate that fulfills this condition "valid". To explain what a valid activity candidate is, we take a closer look at the recognized activity pattern candidate ap_5 in our example (see Figure 2). The ap_5 activity pattern describes the transformation of an association with association class into a model structure consisting of a class and two associations. The transformation can be realized by adding a new class, by transferring all properties of the association class into the new class, by associating the class to both ends (classes) of the original association, and finally by deleting the original association including the connected association class. Figure 2 shows that activity candidate ac_5 has got two allocated events *Add Class* and *Add Association*. Two more editing operations would be required to complete the activity, the deletion of the original association with association class and the creation of another association. Figure 1 shows the final state of our model and it is visible that no association with association class is contained. That means that activity candidate ac_5 cannot be completed, it is invalid and will be filtered.

To realize the filtering we derive model queries from activity candidates that are applied to the repository of the modeling environment. These queries verify the existence of model elements required to complete a partly matched activity. Queries are executed for all activity candidates after each incoming event on the current state of the model. The concrete content of a query depends on the completeness of an activity candidate as each event allocation may add new conditions related to the required model state. Similarly, each editing operation may validate or invalidate existing activity candidates. Accordingly, queries for all activity candidates are derived and executed upon each incoming event.

4.3 Step 3: Ranking Activity Candidates by Relevance

The previous filtering step results in a set of valid activity candidates. In order to present useful recommendations, activity candidates need to be ranked according their relevance for a developer. This ranking requires criteria that are

able to characterize the relevance of an activity candidate. Based on available information about activity candidates, on related work and on our own industrial modeling experiences, we identified three ranking criteria.

α_{ac} – Average age of allocated events of an activity candidate
β_{ac} – Average period between allocated events of an activity candidate
γ_{ac} – Completeness of an activity candidate

We do not claim that these three criteria are the only possible, but we will demonstrate their effectiveness within the evaluation section. It will be a future exercise to further explore the area for other criteria. The following subsections describe each criterion in detail and demonstrate their influence on the ranking of activity candidates. Table 1 shows the ranking results after performing edit operation eo_1 – eo_7 in the prototype. For our example we know the three carried out modeling activities and highlighted them within the columns.

Table 1. Visualization of the influence of the identified ranking criteria on the set of activity candidates $AC(t_0)$ after executing editing operation eo_1 to eo_7 (see Figure 1) in the prototype. The second column shows the order of activity candidates as delivered by the rule engine. The third to fifth column rank these candidates based on a single ranking criterion each. Finally, column six shows the resulting list ranked by combining all three criteria. Cells within the table reflect the activity pattern type ap of a recognized candidate ac and the value calculated for the criterion.

		Ranked based on			
Rank	Non-ranked	α_{ac}	β_{ac}	γ_{ac}	$\alpha_{ac} + \beta_{ac} + \gamma_{ac}$
1	ap_3	ap_3 (1.00)	ap_5 (1.00)	ap_6 (0.75)	ap_5 (0.87)
2	ap_4	ap_4 (1.00)	ap_6 (0.99)	ap_{13} (0.75)	ap_{13} (0.70)
3	ap_{10}	ap_{10} (1.00)	ap_{13} (0.34)	ap_{10} (0.50)	ap_6 (0.69)
4	ap_{13}	ap_{13} (1.00)	ap_3 (0.00)	ap_{17} (0.50)	ap_5 (0.52)
5	ap_{17}	ap_{17} (1.00)	ap_4 (0.00)	ap_{10} (0.50)	ap_{10} (0.38)
6	ap_5	ap_5 (1.00)	ap_{10} (0.00)	ap_{17} (0.50)	ap_{17} (0.38)
7	ap_6	ap_7 (0.98)	ap_{13} (0.00)	ap_5 (0.50)	ap_7 (0.37)
8	ap_3	ap_5 (0.96)	ap_{17} (0.00)	ap_5 (0.50)	ap_3 (0.31)
9	ap_4	ap_{13} (0.71)	ap_5 (0.00)	ap_7 (0.50)	ap_4 (0.31)
...	...	...	...	...	...
11	ap_{13}	ap_3 (0.20)	ap_4 (0.00)	ap_{17} (0.50)	ap_5 (0.31)
...	...	...	...	...	...
15	ap_7	ap_6 (0.03)	ap_7 (0.00)	ap_5 (0.25)	ap_{17} (0.13)
...	...	...	...	...	...
20	ap_{17}	ap_{17} (0.00)	ap_{17} (0.00)	ap_{13} (0.25)	ap_{13} (0.06)
21	ap_5	ap_5 (0.00)	ap_5 (0.00)	ap_5 (0.25)	ap_5 (0.06)

Average age of allocated events. Based on our experience, we assume that a human developer can only work on a limited number of tasks in parallel. Thus, modeling is rather continuous and started modeling activities will be completed within a restricted period of time. Accordingly, it is more likely that a developer is actually working on a younger activity candidate than one that has been recognized a longer time ago. To address this fact, we define the current age of

an event e_y as the time span between the occurrence of the last triggered event t_0 and its own occurrence t_{e_y}:

$$a_{e_y}(t_0) = t_0 - t_{e_y}. \tag{1}$$

The average age of all n allocated events of an activity candidate ac_x is determined as:

$$\bar{a}_{ac_x}(t_0) = \frac{\sum(a_{e_1}, \ldots, a_{e_n})}{n}. \tag{2}$$

Let $A = (\bar{a}_{ac_1}, \ldots, \bar{a}_{ac_m})$ be the set of average ages for all m activity candidates at t_0. The ranking criterion α_{ac} of an activity candidate ac_x is defined as:

$$\alpha_{ac_x}(t_0) = 1 - \frac{\bar{a}_{ac_x} - min(A)}{|max(A) - min(A)|}. \tag{3}$$

The formula means that an increasing average age of an activity candidate decreases its likeliness of being relevant. Values of α_{ac} are normalized on a scale between zero and one. The candidate with the smallest average age receives a value of 1.0, while the one with the largest average age receives a value of 0. This is done to assess proportions between activity candidates rather than absolute values to ensure comparability over time. The influence of α_{ac} on the ranking for the our example is illustrated in Table 1.

Average period between allocated events. Not only the average age but also the period between the allocated events influences the relevance of an activity candidate. We assume that events establishing the same modeling activity more likely occur within a limited period of time. Even if the completion is interrupted, see the phone call in our running example (Section 3), most editing operations are carried out as a contiguous sequence. Thus, an activity candidate is more likely to be irrelevant if its allocated events occurred with long periods in between. To assess a candidate ac_x for this criterion, we calculate the average period of its allocated events as the time span between the first (t_{first}) and the last (t_{last}) allocated event divided by the total number of allocations n:

$$\bar{p}_{ac_x} = \frac{t_{first_x} - t_{last_x}}{n} \tag{4}$$

Let $P = (\bar{p}_{ac_1}, \ldots, \bar{p}_{ac_m})$ be the set of average periods for all m activity candidates at t_0. The ranking criterion β_{ac} of an activity candidate ac_x is defined as:

$$\beta_{ac_x} = 1 - \frac{\bar{p}_{ac_x} - min(P)}{|max(P) - min(P)|} \tag{5}$$

An increasing average period for an activity candidate decreases its likeliness of being relevant. Values of β_{ac} are normalized on a scale between zero and one. The candidate with the smallest average period in the set of current activity candidates receives the value 1.0, while the one with the largest average period receives the value 0. The influence of β_{ac} on the ranking for the example is illustrated in Table 1.

Completeness. Activity candidates are detected by comparing events against specified activity patterns. We assume that the relevance of detected activity candidates increases with each additional allocated event, i.e., with its completeness. The completeness of an activity candidate ac_x is assessed by calculating the ratio of its currently allocated events (n_{alloc}) to the total number of expected editing operations (n_{total}) establishing the corresponding activity:

$$\gamma_{ac_x} = \frac{n_{alloc_x}}{n_{total_x}} \tag{6}$$

The influence of γ_{ac} on the ranking for the example is illustrated in Table 1. Figure 2 shows the completeness of recognized activities and that the activity candidates ap_6 and ap_{13} should be ranked to the top of the list regarding that criterion.

Combining ranking criteria. Finally, the described ranking criteria need to be combined into an overall probability value for each activity candidate. Without history data about the interplay of the identified ranking criteria, a possible way for combining single criteria is to average them, treating each criterion as equally important. The last column of Table 1 shows this probability for our example. However, in order to maximize the quality of recommendations, history data should be used to compute an optimized statistical model that treats the influence of the ranking criteria individually. This approach has been used for the computation of our experimental results in Section 5.

4.4 Step 4: Presenting Recommendations

By filtering (Step 2) and ranking (Step 3), an ordered list of relevant activity candidates has been created. Although, candidates representing relevant activities are ranked topmost, the set likely contains many additional valid but less relevant entries. Reed found [22] that humans can only comprehend a limited number of recommendations effectively. Hence, it is necessary to limit presented recommendations to a useful number. Holmes et al. [9] also identified that need and refer to a list of ten entries as useful. For the computation of our experimental results (see Section 5), we follow that suggestion, but also explore a more sophisticated method that takes into account the overall probability of recommendations.

5 Evaluation

We evaluated our approach according to the challenges C1 to C4 described in Section 1. Challenges C1 and C2 require the ability to recognize partly performed activities, which forms the basis of our approach. Thereby, the system has to handle user variability such as different orders of editing operations to perform the same modeling activity. We evaluated the tolerance of modeling variations of our approach within Experiment 1. A crucial aspect of user acceptance for

a recommendation approach is its performance. We evaluated the performance of our approach with Experiment 2, which refers to challenge C3. Challenge C4 addresses the usefulness of generated recommendations. We evaluated this aspect within the extensive Experiment 3. The extensibility of our approach (Challenge C5) was not evaluated for this paper. We are enhancing the recommender system with auto-completion functionality within the ongoing research work and we will demonstrate its usage on different platforms and with an extended catalog of activity patterns as future work.

5.1 Experimental Setup

We implemented our recommender system as plug-in for the commercial modeling environment Sparx Enterprise Architect [16]. The prototype embeds the Drools 5.5 rule engine. All experiments were performed on a system with an Intel i7 2.7GHz processor, 4GB RAM, and a 64-bit Windows Microsoft 7 OS.

Our experiment is using recorded data sets of an experiment with 16 subjects that performed modeling tasks over a period of approximately two hours each. The experiment was originally conducted to evaluate our *traceMaintainer* approach [19]. In the original experiment we were purely interested in the quality and efficiency of traceability maintenance possible for the modeling tasks performed by a subject . However, we also logged all editing operations performed by the subjects and use that data to evaluate our recommendation approach.

A medium size model-based development project of a mail-order system was used. This project comprised various UML diagrams on three levels of abstraction: requirements, design, and implementation. Subjects had to perform three maintenance tasks on the mail-order system. First, the system's functionality had to be enhanced to distinguish private and business customers and to handle foreign suppliers. Second, the system layers view and data had to be extracted into separate components. And third, the system's functionality had to be enhanced to handle additional product groups and to categorize products according to content categories. Tasks were described in general terms in order to acquire a wide spread of different solutions. The experiment was performed by 16 computer science students that were either in the fourth or fifth year of their university studies. All students were taking a course on software quality and had advanced experience in model-based software engineering and UML. The 16 acquired data sets contained recorded events describing each performed editing operation. As we kept the event notation consistent with our previous approach, data could directly be used for our evaluation. All experimental material is available in [18].

5.2 Experiment 1: Modeling Variance Toleration

Experiment 1 was conducted to evaluate the toleration by the recognition part of our approach for modeling variances in performing the same activity pattern. The approach must recognize the same activity patterns within different permutations of the same event sequences. Therefore, we replayed 100 randomly generated permutations of the recorded events and compared the computed

recognition result with the original recognitions. To validate alternative ways for executing activities, the experiment was conducted for all 16 different subjects. All generated experiment results contained exactly the same set of activity recognitions as their corresponding original, but in different orders according to the permutation of events.

5.3 Experiment 2: Performance

We measured the execution time across all processing steps, i.e., from the occurrence of an event triggered by an editing operation to the fully presented recommendation set within Sparx Enterprise Architect. This time is independent of the model size but depends on the number of processed events and the number of activity patterns defined in the catalog. We computed the average execution time $\bar{t}_x$ and the maximum execution time t_{max} across all performed editing operations of the 16 subjects for a number of 5, 10 and 19 defined patterns. Results are discussed in Section 6:

$$t_{mean} = \frac{\bar{t}_1, \ldots, \bar{t}_{16}}{16}, t_{mean_{5,10,19}} = (143ms, XXXms, XXXms), \\ t_{max} = max(t_{m1}, \ldots, t_{m16}) = 197ms \tag{7}$$

5.4 Experiment 3: Relevance of Recommendations

In this experiment we evaluated the relevance of generated recommendations for the user. First, we determined for each editing operation that a subject had performed all activity candidates that she/he started at this point and that she/he completed during the remaining modeling session. These identified activity candidates comprised a golden master of relevant activities per editing operation of a subject. We compared this golden master with the actual recommendations of our approach and evaluated the relevance of recommendations with the averaged common metrics recall, precision, and average precision. Mean recall (MR) measures shown relevant recommendations in relation to all relevant recommendations across all editing operations made by a subject. Mean precision (MP) measures the amount of relevant recommendations in relation to all shown recommendations across all editing operations made by a subject. Finally, mean average precision (MAP) measures the precision of recommendations at every position in the ranked sequence of recommendations across all editing operations of a subject. This metric evaluates the ranking performance of our approach.

We determined an optimized weighting function for the three ranking criteria by performing a binominal regression based on our evaluation data (see Section 4.3). Due to the limited amount of subjects performing the experiment, we applied a leave-one-out cross-validation strategy [23] across the acquired 16 data sets. We fitted 16 generalized linear models, every time using 15 out of 16 data sets. These models were then used to rank the recommendations of the 16th data set. Results are shown in Table 2. We applied three threshold strategies to cut the list of available recommendations to a comprehensible number (see Section

Table 2. Relevance of computed recommendations for all subjects performing the experiment measured as mean recall (MR), mean precision (MP), and mean average precision (MAP) at three different thresholds (th)

	$th_{max} = 10$ $th_{prob} = /$			$th_{max} = /$ $th_{prob} = 0.02$			$th_{max} = 10$ $th_{prob} = 0.02$		
Subject	MAP	MR	MP	MAP	MR	MP	MAP	MR	MP
1	100.00%	100.00%	13.40%	100.00%	100.00%	37.08%	100.00%	100.00%	37.08%
2	71.61%	98.82%	34.92%	71.84%	100.00%	33.51%	71.61%	98.82%	37.11%
3	65.55%	100.00%	48.06%	67.88%	89.56%	48.40%	67.88%	89.56%	48.40%
4	56.47%	62.02%	15.74%	45.55%	80.05%	16.42%	56.47%	62.02%	18.42%
5	55.59%	76.69%	11.83%	50.13%	86.72%	10.20%	55.59%	76.69%	12.19%
6	60.72%	99.34%	21.46%	65.33%	83.11%	15.19%	65.33%	83.11%	19.92%
7	82.15%	92.58%	12.83%	79.32%	93.75%	11.94%	82.12%	90.23%	14.30%
8	71.94%	100.00%	10.00%	71.94%	100.00%	10.10%	71.94%	100.00%	12.87%
9	60.45%	64.89%	17.31%	59.00%	77.43%	24.75%	66.45%	54.50%	22.88%
10	77.26%	100.00%	15.11%	77.26%	100.00%	17.74%	77.26%	100.00%	18.38%
11	77.40%	100.00%	12.14%	77.70%	98.60%	10.98%	77.70%	98.60%	14.78%
12	57.32%	52.87%	17.46%	49.24%	59.62%	25.15%	76.96%	31.20%	23.83%
13	89.84%	100.00%	17.81%	89.52%	96.88%	21.37%	89.52%	96.88%	25.91%
14	48.12%	61.22%	16.67%	41.20%	83.84%	14.04%	48.12%	61.22%	17.17%
15	97.74%	100.00%	16.76%	97.74%	100.00%	15.42%	97.74%	100.00%	18.78%
16	82.93%	100.00%	17.03%	82.93%	100.00%	24.41%	82.93%	100.00%	25.54%
Average	72.19%	88.03%	18.66%	70.41%	90.60%	21.04%	74.23%	83.93%	22.97%

4.4). Columns 2–4 show the three metrics for a fixed cutoff of 10 recommendations. Columns 5–7 show the metrics for a dynamic cutoff at probability 0.02. Finally, columns 8–10 show the metrics for a cutoff at probability 0.02 or at 10 recommendations, whatever occurs earlier.

6 Discussion

In Experiment 1, we evaluated the tolerance of the approach for possible variations in a developer's flow of editing operations. We found that we recognized the same set of modeling activities across 100 permutations of the editing operations performed by each subject. This result shows that our event processing and activity pattern matching implementation is independent of the order of events and tolerates variances in the way a developer performs a modeling activity.

In Experiment 2, we studied the performance of the approach and found that the generation of recommendations after an editing operation consumed on average 143ms with a maximum of 197ms for a set of 19 activity patterns. We consider these values as unintrusive for a user. However, computation time of the activity pattern matching depends on the number of patterns defined in the catalog and on the number of events kept in the matching process. The results show that computation time rises linearly with the number of defined patterns. Our subjects performed on average 219 editing operations during the experiment. Drool's implementation is known as very efficient and the algorithm itself as the state of the art for complex event processing. However, it might be necessary to limit the event history for models with many editing operations in order

to guarantee a certain computation time. We are planning a more substantial performance evaluation regarding those facts as part of a future industrial study.

In Experiment 3, we studied the relevance of recommendations generated by our approach (see Table 2). Focusing on the first threshold strategy, which always presents the ten most relevant recommendations to the user, we recommended across all 16 subjects 88% (MR) of the activities that a user was actually working on. These relevant recommendations comprised 19% (MP) of all recommendations. The value of 72% for the MAP metric shows that we rank relevant recommendations close to the top of the list. All three values are very promising and show that we were able to recommend the majority of activities performed by the user in a list of ten elements. The other two thresholding strategies show similar promising results. Whether these results are good enough to get acceptance is a research question for ongoing work, which will be evaluated once the whole auto-completion approach is available.

Concluding, the results of our evaluation show that the proposed approach meets the challenges 1–4 discussed in Section 1. However, our evaluation is limited in several regards. The studied modeling activities were carried out over a relatively short period of two hours and all subjects were solving the same tasks. However, this experiment ensured that we captured manifold editing operation sequences with similar goal and evaluated the tolerance for developer variances. The computed regression models that combined individual ranking criteria are based on the data of other subjects performing the the same modeling tasks, this approach might have biased our results positively. We clearly identify the need for more empirical and industrial evaluation to draw general conclusions about the applicability of our approach.

7 Conclusions and Future Work

We presented an approach for recommending auto-completions of modeling activities performed on structural UML-models. We identified five challenges that had to be handled for making a recommendation approach useful to a user. The developed approach addresses these challenges. It works in the background while a developer is creating or evolving a model. Editing operations are analyzed and matched to a predefined but extensible catalog of common modeling activities for structural UML models. We evaluated our approach in a controlled experiment with 16 students evolving models. We recommended 88% of the activities that the subjects wanted to perform within a short list of ten recommendations.

We are currently working on an auto-completion mechanism for selected recommendations to complement our approach. Once both approaches are available, we are planning an industrial study to gain more empirical data on performed modeling activities, user preferences, and the discussed ranking criteria.

Acknowledgment

We are supported by the German Research Foundation (DFG): Ph49/8-1 and the German Ministry of Education and Research (BMBF): Grant No. 16V0116.

References

1. Arlow, J., Neustadt, I.: UML and the unified process: practical object-oriented analysis and design. 2 edn. Number ISBN 0-321-32127-8. Addison-Wesley (2006)
2. Fowler, M.: Refactoring: improving the design of existing code. 19 edn. Number ISBN 0-201-48567-2. Addison-Wesley (2006)
3. University of Illinois at Chicago: Optimizing the object design model: Course notes for object-oriented software engineering. `http://www.cs.uic.edu/~jbell/CourseNotes/OO_SoftwareEngineering/MappingModels.html` (Accessed: 15/03/2013).
4. Sun, Y., White, J., Gray, J.: Model transformation by demonstration. In Schürr, A., Selic, B., eds.: Model Driven Engineering Languages and Systems. Volume 5795 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2009) 712–726
5. Sun, Y., Gray, J., Wienands, C., Golm, M., White, J.: A demonstration-based approach to support live transformations in a model editor. In Cabot, J., Visser, E., eds.: Theory and Practice of Model Transformations. Volume 6707 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2011) 213–227
6. Ráth, I., Bergmann, G., Ökrös, A., Varró, D.: Live model transformations driven by incremental pattern matching. In Vallecillo, A., Gray, J., Pierantonio, A., eds.: Theory and Practice of Model Transformations. Volume 5063 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2008) 107–121
7. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence **19**(1) (1982) 17 – 37
8. Murphy-Hill, E., Jiresal, R., Murphy, G.C.: Improving software developers' fluency by recommending development environment commands. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. FSE '12, New York, NY, USA, ACM (2012) 42:1–42:11
9. Holmes, R., Walker, R., Murphy, G.: Approximate structural context matching: An approach to recommend relevant examples. Software Engineering, IEEE Transactions on **32**(12) (2006) 952–970
10. Foster, S.R., Griswold, W.G., Lerner, S.: Witchdoctor: Ide support for real-time auto-completion of refactorings. In: Software Engineering (ICSE), 2012 34th International Conference on. ICSE 2012, Piscataway, NJ, USA, IEEE Press (2012) 222–232
11. Mazanek, S., Maier, S., Minas, M.: Auto-completion for diagram editors based on graph grammars. In: Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008. IEEE Symposium on. (2008) 242–245
12. Mazanek, S., Minas, M.: Business process models as a showcase for syntax-based assistance in diagram editors. In Schürr, A., Selic, B., eds.: Model Driven Engineering Languages and Systems. Volume 5795 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2009) 322–336
13. Sen, S., Baudry, B., Vangheluwe, H.: Towards domain-specific model editors with automatic model completion. Simulation **86**(2) (2010) 109–126

14. Reder, A., Egyed, A.: Computing repair trees for resolving inconsistencies in design models. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. ASE 2012, New York, NY, USA, ACM (2012) 220–229
15. Steimann, F., Ulke, B.: Generic model assist. In: MODELS 2013-16th International Conference on Model Driven Engineering Languages and Systems. (2013)
16. Sparx Systems: Enterprise architect: A model driven uml tool suite. `http://www.sparxsystems.com` (Accessed: 15/03/2013).
17. IBM: Rational software architect: Colaborative systems and software design. `http://www-01.ibm.com/software/rational/products/swarchitect` (Accessed: 15/03/2013).
18. Mäder, P.: Rule-based maintenance of post-requirements traceability. PhD thesis (2010)
19. Mäder, P., Gotel, O.: Towards automated traceability maintenance. Journal of Systems and Software **85**(10) (2012) 2205 – 2227
20. Paschke, A., Kozlenkov, A.: Rule-based event processing and reaction rules. In Governatori, G., Hall, J., Paschke, A., eds.: Rule Interchange and Applications. Volume 5858 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2009) 53–66
21. Red Hat: Drools 5: An integrated platform for rules, workflows and event processing. `http://www.jboss.org/drools` (Accessed: 15/03/2013).
22. Reed, A.V.: List length and the time course of recognition in immediate memory. Memory & Cognition **4**(1) (1976) 16–30
23. Arlot, S., Celisse, A.: A survey of cross-validation procedures for model selection. Statistics Surveys **4** (2010) 40–79