

A Demonstration-based Approach to Support Live Transformations in a Model Editor

Yu Sun¹, Jeff Gray², Christoph Wienands³, Michael Golm³, Jules White⁴

¹University of Alabama at Birmingham, Birmingham AL 35294
yusun@cis.uab.edu

²University of Alabama, Tuscaloosa, AL 35401
gray@cs.ua.edu

³Siemens Corporate Research, Princeton, NJ 08540
{christoph.wienands, michael.golm}@siemens.com

⁴Virginia Tech, Blacksburg, VA 24060
julesw@vt.edu

Abstract. Complex model editing activities are frequently performed to realize various model evolution tasks (e.g., model scalability, weaving aspects into models, and model refactoring). In order to automate and reuse patterns of model editing, an editing process can be regarded as an endogenous model transformation and specified as transformation rules. However, the use of traditional model transformation languages often presents a steep learning curve. Other challenges in using model transformations to automate editing tasks include insufficient support for sharing the transformations that perform the editing tasks, and a lack of automated guidance on how to use a specific transformation in some other modeling context. This paper presents a live model transformation approach that can enhance and assist model editing activities. By extending the Model Transformation By Demonstration (MTBD) approach, LiveMTBD offers users flexibility in specifying the transformation, a centralized repository to assist with transformation sharing, and a live model transformation matching engine to suggest applicable transformations during model-edit time.

Keywords: Model Editing, Live Model Transformation By Demonstration.

1 Introduction

With the ongoing adoption of Model-Driven Engineering (MDE) [1], models are emerging as first-class entities in many domains and play an increasingly significant role in every phase of software development. During the process of building models, editing operations are constantly performed in the editor to create or change the model into the desired state and configuration. For instance, a sequence of creational operations is needed to construct a typical sub-structure in a certain domain; model refactoring actions might be required occasionally to optimize the internal structure of the represented system; when errors are detected in models, operations to fix errors should be carried out in a timely

manner; and with non-functional system requirements, aspect models may need to be woven into the desired locations of the base models.

The editing activities mentioned above often integrate ideas of model evolution, which involve composite editing operations on specific locations, and are very likely to be repeated in different model instances by different users. Therefore, a mechanism to automate and reuse frequently used editing patterns can benefit the model editing process. One approach to support automation and reuse of editing activities is to apply model transformation techniques. Any editing operation performed in the editor will change specific model instances, which can be considered as an endogenous model transformation process [9]. Thus, the sequence of editing operations for certain purposes can be summarized and specified as a set of transformation rules using executable Model Transformation Languages (MTLs) [4]. The rules may be directly reused in other model instances in the same domain, such that executing the rules triggers the desired editing activity automatically. However, although traditional MTLs are very powerful and expressive for specifying many editing activities, several challenges have emerged in using MTLs that prevent it from being a perfect solution:

Challenge 1. The steep learning curve of model transformation languages prevent general end-users (e.g., domain experts, non-programmers) from contributing to the editing or evolution tasks from which they have much domain experience.

Challenge 2. The abstraction gap between the concrete editing operations and the metamodel level transformation rules make the specification of the desired editing activity challenging and perhaps even impossible for certain classes of end-users.

Challenge 3. Most MTLs and their supporting tools lack a collaborative mechanism to enable the sharing of the transformation rules among different model users, limiting the opportunity for reuse and exchange of domain-specific editing patterns.

Challenge 4. Without correctly understanding the transformation rules at the metamodel level or knowing the existence of the rules, novice users might miss the correct situations on when to reuse a specified editing activity.

To address these challenges, this paper presents an enhanced demonstration-based model transformation approach – Live Model Transformation By Demonstration (LiveMTBD). The idea is an extension to the Model Transformation By Demonstration (MTBD) [2] approach, which was designed to simplify the implementation of model transformation tasks by inferring and generating model transformation patterns from user-demonstrated behavior. The goal of MTBD is to enable general end-users in realizing their desired model transformation tasks without knowing a model transformation language or metamodel definitions. Applying MTBD to support automatic reuse of editing activities can assist end-users in avoiding the steep learning curve of MTLs and abstract metamodel definitions. By extending MTBD, LiveMTBD contains three new features: 1) *Live Demonstration*, provides a more general demonstration environment that allows users to specify editing activities based on their editing history; 2) in order to improve the sharing of editing activity knowledge among different users, *Live Sharing* – a centralized model transformation pattern repository has been built so that transformation patterns can be reused across different editors more efficiently; 3) a live model transformation

matching engine – *Live Matching* has been developed to automatically match the saved transformation patterns at modeling time, and provides editing suggestions and guidance to users during the editing process.

The remainder of the paper is organized as follows: Section 2 explains the motivation and challenges with concrete examples borrowed from an industrial context; Section 3 presents the solution by introducing the initial work on MTBD, followed by highlighting new extensions; Section 4 discusses the advantages and limitations of the solution; Section 5 compares the related techniques, and Section 6 offers concluding remarks.

2 Motivating Example

This section overviews the challenges of specifying, reusing and automating model editing activities using MTLs. The examples used in this paper are based on the Embedded Function Modeling Language (EmFuncML), which has been used to support modeling embedded control in automotive industry. EmFuncML enables the following: 1) model the internal computation process and data flow within functions; 2) model the high-level assignment and configurations between functions and supporting hardware devices; 3) generate platform-dependent implementation code; and 4) estimate the Worst Case Execution Time (WCET) for each function.

The top of Figure 1 shows an excerpt of the model describing functions used in an automotive system. *ReadAcc* (i.e., Read Acceleration) reads output data from *ADC* (i.e. Analog-to-Digital Converter) and sends the processed data to the *Analysis* function, which then transmits messages to the *Display* function. The input/output ports of each function are given (e.g., *ADC* has four input ports: *Resolution*, *SamplingRate*, *Downsampling*, *InterruptID*; and one output port *AnalogValue*). The hardware devices (e.g., *ADC*, *ECU*) are presented, to which the corresponding functions are assigned. A tool has been developed to estimate the WCET of each function based on the internal computation logic. For the sake of ensuring a smooth data flow and quick processing time, the WCET of each function should be less than *300ms*; otherwise it is defined as an occurrence of a WCET violation.

One common practice occurring in the configuration of functions in EmFuncML is that if a WCET violation happens, a *Buffering* function can be added between the source function and the target function that receives data to ensure the correct data flow. At the bottom of Figure 1, *Analysis* sends message data to *Display*. However, the WCET of *Analysis* is *460ms*, which is longer than the desired processing time. Therefore, a *Buffering* function is added between *Analysis* and *Display*, which serves as an intermediate storage for the transmitted data.

Embedded software system engineers who are familiar with functional timing requirements may perform the *Buffering* editing activity in the editor very often whenever the WCET violation is detected. Therefore, this process can benefit from automation and reuse, which can be realized using MTLs to specify and summarize the result of the editing activity. Figure 2 shows the pseudo code of the transformation rules to accomplish

the task of applying the *Buffering* function by locating functions with WCET violation, creating *Buffering* function and rerouting the data flow.

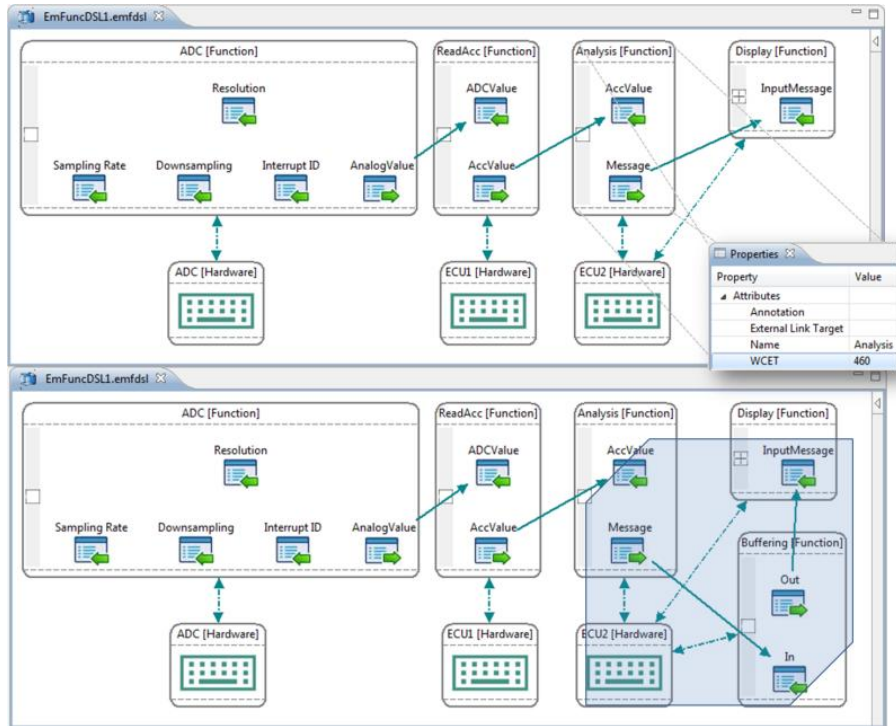


Figure 1. EmFuncML models before (top) and after (bottom) applying *Buffering* function

A number of MTLs and tools can be applied to implement the actual task, such as ATL [3], Epsilon [21], and C-SAW [5], which support expressive mechanisms to access and manipulate models. However, using these languages requires users to learn the syntax and execution semantics, as well as some additional concepts (e.g., OCL is often used to locate model elements) and libraries. In addition, because MTLs operate at the metamodel level, in order to summarize a specific editing activity, users need to think about the whole editing process at a more abstract level and then generalize it using correct metamodel definitions. Sometimes, the difference in the execution semantics of a MTL may lead to different implementation designs (e.g., imperative textual MTLs focus on specific model manipulate steps, while declarative graphical MTLs consider the editing change as a pair of source and target graphs), which requires extra consideration for users in the specification progress.

In addition to the learning curve problem and abstraction gap when using MTLs, sharing the specified model transformations has not been taken into consideration in most MTLs and tools. For example, the left part of Figure 1 shows the *ADC* configuration, which is modeled through a sequence of approximately 20 editing operations to create the

ADC function, input/output ports, set their names and types, and create the *ADC* hardware device with the assignment connection. Hardware engineers are more experienced than software engineers in this part of configuration. Thus, the complex editing operation of creating an *ADC* can be specified as a reusable model transformation by hardware engineers that can be used by different colleagues in their modeling process when the *ADC* needs to be modeled in other system contexts. Clearly, if model transformations can be shared among users with different expertise or levels of experience, the reuse captured in a transformation rule can contribute to a knowledge base, improving the collaborative construction of models in the same domain.

```
foreach Output2Input connection : c
  if (c.source.parent.WCET > 300)
    create Function in EmFuncDSLFolder : tempFunc
    set tempFunc.name = "Buffering"
    create InputPort in tempFunc : tempInput
    set tempInput.name = "In"
    set tempInput.type = c.source.type
    create OutputPort in tempFunc : tempOutput
    set tempOutput.name = "Out"
    set tempOutput.type = c.target.type
    create Output2Input connection : c1 from c.source to tempInput
    create Output2Input connection : c2 from tempOutput to c.target
    create Func2HW connection : c3
      from tempFunc to c.source.parent.mappedHW
    remove Output2Input connection : c
```

Figure 2. The pseudo code to specify *ApplyBuffer* editing activity

Finally, archiving model transformation rules does not guarantee the appropriate and correct reuse of the rules, due to a lack of suggestion or guidance about when and where to apply the transformation rules, particularly when the rules are specified by other users. For instance, it is likely that hardware engineers fail to reuse the *ApplyBuffer* transformation although it has been specified by software engineers, because they do not realize the issues involving WCET. Likewise, when software engineers are trying to configure the correct *ADC* for their system, the *ADC* creation transformation specified by hardware engineers may not be reused either, simply because the software engineers are not aware of the existence of a model transformation that can fulfill their needs directly.

Thus, the contribution of this paper focuses on providing an approach to improve specifying, reusing and automating the model editing activities.

3 Solution: LiveMTBD

Our solution to improve specification, reuse and automation of editing activities is to use a demonstration-based technique with three “live” features (i.e., active processes that monitor and respond to editing activities). In this section, we introduce MTBD in Section 3.1, and then explain the extended new version LiveMTBD in Section 3.2.

3.1 Introduction to MTBD

The basic idea of MTBD is that rather than manually writing model transformation rules, users are asked to use concrete model instances and demonstrate how to transform a source model to a target model by directly editing and changing it. A recording and inference engine captures all of the user operations and automatically infers a transformation pattern that summarizes the changing process. This generated pattern can be executed in any model instance under similar circumstances to repeatedly carry out the desired transformation process. Figure 3 shows the overview of MTBD consisting of the following steps (the components with shaded background are new extensions to LiveMTBD that will be presented in Section 3.2).

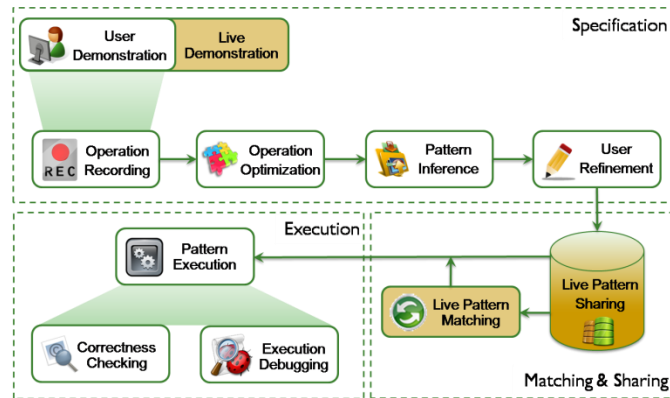


Figure 3. Overview of LiveMTBD (components with shaded background are new extensions to MTBD) (adapted from [2])

Step 1 – User Demonstration and Operation Recording. MTBD starts from a user demonstration about the model transformation process. A desired part of a model instance is located first as the source model, after which users perform basic editing operations (e.g., add a new model element, update its attributes) to change it into the desired target model. A recording engine stores all of the operations occurring in the editor, and saves the context information for all model elements and connections involved. We illustrate the MTBD idea using the motivating example *ApplyBuffer*. On the source model shown in the top of Figure 1, users can demonstrate the process of adding the new *Buffering* function and reconnecting the data flow. List 1 shows the operations performed to complete the demonstration. The bottom of Figure 1 shows the model after the demonstration.

Step 2 – Operation Optimization. Meaningless operations are occasionally present due to a careless demonstration by the user (e.g., add one element and later remove it without using it in between). An algorithm [2] has been designed to eliminate meaningless operations. The operations in List 1 are all meaningful.

Step 3 – Pattern Inference. In MTBD, the transformation process is specified and formalized as a transformation pattern, which is a 2-tuple $\langle P, T \rangle$, where P is the precondition of the transformation specifying where to apply the transformation, and T is

a sequence of transformation actions specifying how the transformation is done. Based on the optimized list of operations, an initial transformation pattern can be inferred, by summarizing all the involved model elements and connections in the demonstration and generalizing their meta types and relationships. The precondition P inferred from this step specifies the minimum structural constraints where the transformation can generally be applied, and the actions T composes of all the operations from the optimized list.

List 1. Operations performed to demonstrate *ApplyBuffer*

Sequence	Operation Performed
1	Add a <i>Function</i>
2	Set <i>Function.name</i> = “ <i>Buffering</i> ”
3	Add an <i>InputPort</i> in <i>Buffering</i>
4	Set <i>InputPort.name</i> = “ <i>In</i> ”
5	Set <i>InputPort.type</i> = <i>Analysis.Message.type</i> = “ <i>string</i> ”
6	Add an <i>OutputPort</i> in <i>Buffering</i>
7	Set <i>OutputPort.name</i> = “ <i>Out</i> ”
8	Set <i>OutputPort.type</i> = <i>Display.InputMessage.type</i> = “ <i>string</i> ”
9	Connect <i>Analysis.Message</i> to <i>Buffering.In</i>
10	Connect <i>Buffering.Out</i> to <i>Display.InputMessage</i>
11	Disconnect <i>Analysis.Message</i> to <i>Display.InputMessage</i>
12	Connect <i>Buffering</i> to <i>ECU2</i>

The initial precondition shown in Figure 4 is inferred from the operations list. It specifies that two connected functions, plus the hardware, must exist to ensure that the recorded operations can be executed with correct and sufficient operands. Then, the transformation actions are generalized operations based on the precondition.

Precondition	Precondition'	Actions
		<ol style="list-style-type: none"> 1. Add Function <i>newFunc</i> 2. Set <i>newFunc.name</i> = “<i>Buffer</i>” 3. Add InputPort <i>newIP</i> 4. Set <i>newIP.name</i> = “<i>IN</i>” 5. Set <i>newIP.type</i> = <i>op1.type</i> 6. Add OutputPort <i>newOP</i> 7. Set <i>newOP.name</i> = “<i>OUT</i>” 8. Set <i>newOP.type</i> = <i>ip1.type</i> 9. Connect <i>op1</i> to <i>newIN</i> 10. Connect <i>newOP</i> to <i>ip1</i> 11. Connect <i>newFunc</i> to <i>h1</i> 12. Remove <i>c1</i>

Figure 4. Model transformation pattern after Step 3 (Precondition) and Step 4 (Precondition')

Step 4 – User Refinement. The initially inferred transformation pattern is sometimes not generic and accurate enough due to the limitations of the expressiveness of a user demonstration. For instance, the precondition P only reflects the structural constraints on the elements that are touched in the demonstration, ignoring the elements that were not directly edited in the demonstration as well as the attribute constraints. From Figure 4, it can be seen that the required assignment connection between the function $f1$ and the

hardware hl is missing, and the constraint on WCET is not specified. Thus, users can make refinements by either confirming more elements and connections to the structural precondition or specifying detailed constraints on the attribute precondition. In addition, user refinement can also be performed on the transformation actions to identify the generic operations, which should be executed repeatedly according to the actual number of available model elements and connections. The finalized transformation pattern after user refinement $\langle P', T' \rangle$ is stored in the repository for future reuse.

In our example, two additional operations in List 2 are carried out to refine the initial transformation pattern. The containment relationship is simply done by clicking on the desired connection and confirming its existence in the pattern. The attribute precondition is given through a dialog where users can choose any model elements and connections touched in the demonstration and specify the needed constraint expressions. No refinement is performed on actions in this case. The finalized pattern is shown in Figure 4.

List 2. Operations performed for *ApplyBuffer* in the demonstration

Sequence	Operation Performed
13	Confirm the containment of assignment between <i>Analysis</i> and <i>ECU2</i>
14	Add an attribute constraint on <i>Analysis</i> – <i>Analysis.WCET > 300</i>

Step 5 – Pattern Execution. The transformation patterns can be reused in any model instances at any time. The execution process can be formalized as a function with two parameters: *EXECUTION* ($\langle P', T' \rangle, I$), where $\langle P', T' \rangle$ is a finalized transformation pattern, and I is the input candidate pool of model elements and connections to match the pattern. The execution process starts by matching precondition P' in the candidate pool I , followed by executing transformation actions T' in the matched locations. A back-tracking algorithm [2] has been implemented to realize the matching, and the execution of transformation actions is completed using model manipulation APIs. Users can customize the input candidate pool by either using the default full selection (all model elements and connections in the editor) or choosing specific model elements or connections. The execution of *ApplyBuffer* pattern will match all the function pairs based on the precondition and execute the actions to reconnect them through a *Buffering* function.

Step 6 – Correctness Checking and Debugging. Because the precondition P' does not ensure that the execution will not violate the syntax, semantics definitions or external constraints, the execution of each transformation action will be logged and the model instance correctness checking is performed after every execution. Whenever a violation occurs, all executed actions are undone and the whole transformation is cancelled. A high-level debugger is under development to enable end-users to track the execution of the transformation pattern and prevent abstraction leaks.

3.2 From MTBD to LiveMTBD

LiveMTBD consists of three new components as shown in Figure 3. The contributions of LiveMTBD include new capabilities that improve the specification, sharing, and reuse of model transformation patterns within the MTBD framework.

Live Demonstration. Although the specification of model transformation patterns using MTBD does not require the use of MTLs or the knowledge of metamodel definition, users must plan ahead and provide explicitly a demonstration that specifies the desired editing activity. A challenge is when a user does not realize the potential for reusing an editing activity until it is part-way through. For example, the hardware engineer configures *ADC* by performing a sequence of editing operations. After the editing is completed, the engineer may then think (post-editing) that because the *ADC* is a commonly used component in embedded systems, the editing activity just performed should be summarized and saved as a reusable model transformation pattern. Therefore, he or she may begin a demonstration and repeat exactly the same editing operations for the sake of inferring the transformation pattern. This repetition could be tedious and time-consuming if the editing activity to demonstrate is complex.

In order to enable a more flexible demonstration approach, live demonstration is implemented so that the recording engine works continuously to record every editing operation performed in the editor. Then, whenever a user realizes a need to specify and summarize a certain model transformation pattern for an editing activity, they can simply go back to the recording view and check all the operations that are related with the specific editing activity, after which the original MTBD inference engine infers the transformation from the archived editing events. In this way, users specify their desired editing activity by reflecting on their editing history, rather than by an intentional demonstration.

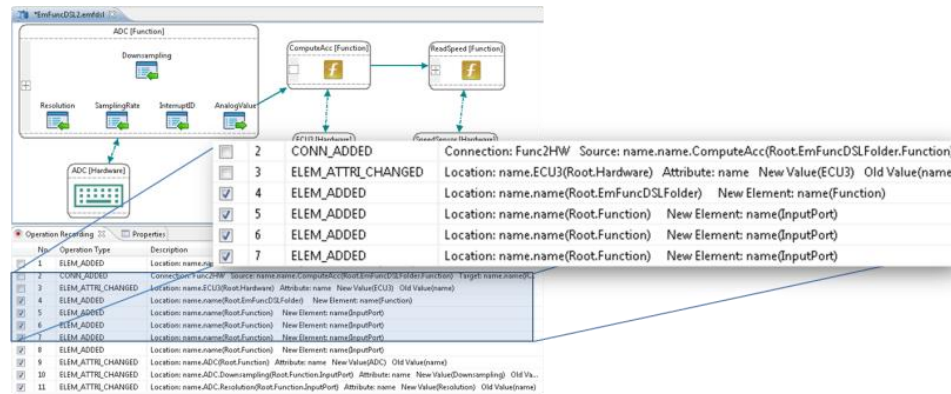


Figure 5. Live demonstration enables demonstration by checking the editing history

As can be seen in the example from Figure 5, a user creates the whole model by adding the *ComputeAcc* function, *ADC* function and hardware, and then *ReadSpeed*. After the complete model is specified, the user may check the related operations from the recording view and then generate the transformation pattern (e.g., the *CreateADC* transformation as shown in Figure 6). This pattern can be applied to any function, and changes the selected function into a fully configured *ADC* function by adding four input ports and one output port, as well as the corresponding *ADC* hardware.


Precondition	Actions	
	<ol style="list-style-type: none"> 1. Set <i>f1.name</i> = "ADC" 2. Add InputPort <i>ip1</i> 3. Set <i>ip1.name</i> = "Resolution" 4. Set <i>ip1.type</i> = "double" 5. Add InputPort <i>ip2</i> 6. Set <i>ip2.name</i> = "Downsampling" 7. Set <i>ip2.type</i> = "double" 8. Add InputPort <i>ip3</i> 9. Set <i>ip3.name</i> = "SampingRate" 10. Set <i>ip3.type</i> = "double" 	<ol style="list-style-type: none"> 11. Add InputPort <i>ip4</i> 12. Set <i>ip4.name</i> = "InterruptID" 13. Set <i>ip4.type</i> = "String" 14. Add OutputPort <i>op1</i> 15. Set <i>op1.name</i> = "AnalogValue" 16. Set <i>op1.type</i> = "double" 17. Add Hardware <i>h1</i> 18. Set <i>h1.name</i> = "ADC" 19. Connect <i>f1</i> to <i>h1</i>

Figure 6. Final transformation pattern for *CreateADC*

Live Sharing. The original MTBD saves finalized patterns locally. To ease the sharing of patterns and enhance the editing activities, LiveMTBD changes the repository to a centralized repository, which can be accessed by any user at any time. The original transformation patterns are persisted as objects. The centralized repository is implemented using Java RMI, which makes the transmission of pattern objects simple and transparent.

With the patterns being stored automatically in the centralized pattern repository, they are available for all users to choose in the pattern execution step, which provides a live collaborative environment. With this feature, various categories of end-users (e.g., software engineers and hardware engineers) can exchange and benefit from each other's knowledge in model editing.

Live Matching. Without a full understanding of all the model transformation patterns, users might miss reusing the correct transformation in the appropriate situation. Although executing all the transformation patterns can automatically match the applicable editing activities, it is very expensive to restore the model if some patterns change the model into an undesired configuration state. To address the problem, live matching in LiveMTBD offers users guidance about applicable model transformation patterns during the editing. Live matching is a function that takes two input parameters: $MATCH(R, I)$, where R is the set of all available model transformation patterns $\langle P', T' \rangle$ in the centralized repository, and I is from the user-selected input candidate pool of model elements and connections. Similar to pattern execution (Step 5), I includes all the model elements and connections in the current editor by default, or a sub-part of the model based on a user's selection. The function returns all the patterns that their precondition P' can be satisfied in I , as well as the number of matched locations. Different from the pattern execution, live matching does not execute the pattern until a user's approval.

To enable live matching, the $MATCH$ function is triggered during two occasions: 1) the selected input model candidate pool I changes, or 2) the available pattern set R in the repository changes.

As an example shown in the top of Figure 7, after we finalize the two transformation patterns – *CreateADC* and *ApplyBuffer*, if the users do not select any part of the model, the whole model instance is included in I , and live matching indicates that both patterns can be applied. Because there are five functions available in the current editor, *CreateADC* is matched 5 times; while the *ApplyBuffer* can be matched to the *ReadSpeed*

function whose WCET is greater than 300. Double-clicking on any of the matched patterns triggers its execution directly.

At the bottom of Figure 7, a user may change the selections on the model from the default to the single function newly added to the model. At this point, only *CreateADC* can be matched, and the precondition of *ApplyBuffer* cannot be satisfied due to the insufficient model elements and connections in the input candidate pool. Executing *CreateADC* can automatically transform this function to a fully configured *ADC* function.

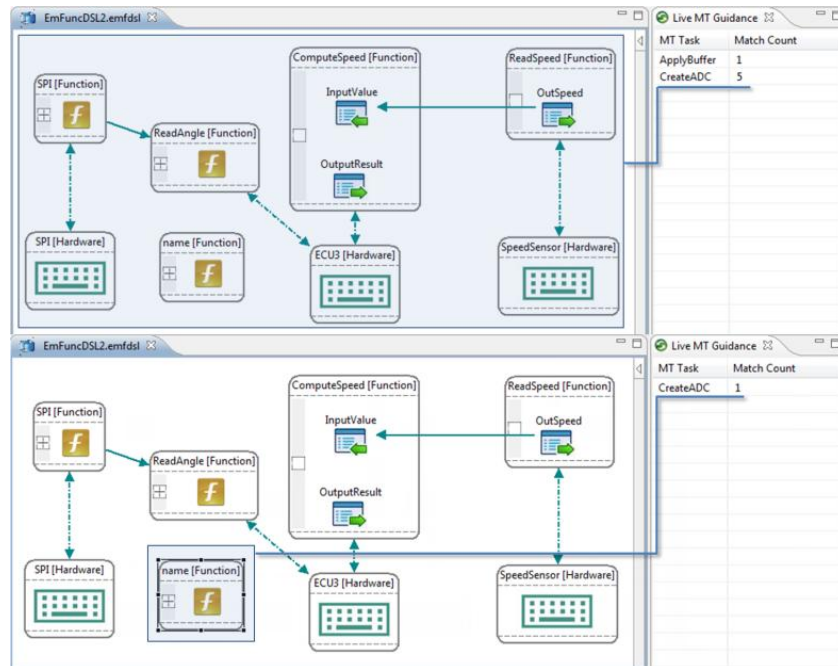


Figure 7. Live matching suggests applicable transformations in the current selection

4 Discussion

LiveMTBD has been implemented by extending the original MTBD tool, which is a plugin to the Generic Eclipse Modeling Systems (GEMS) [20]. Based on the challenges identified in Section 1, it can be seen that LiveMTBD offers the following advantages.

Simplified specification of desired editing activities. In LiveMTBD, no model transformation languages and tools are used in the process, so that users are completely isolated from the need to know and learn MTLs. The only steps that a user is involved are demonstrating the editing process (Step 1) and making refinements (Step 4). All of the other procedures (i.e., optimization, inference, generation, execution, and correctness

checking) are fully automated. In addition, information exposed to users is at the model instance level in the editor, rather than the metamodel level. The generated patterns are invisible to users (Figure 4 and 6 are presented for the sake of explanation, which are not visible to users when using LiveMTBD). Therefore, users are prevented from knowing metamodel definitions and implementation details. With the live demonstration feature, the specification of model transformation patterns can be realized at any moment of the editing task by reflecting and checking the editing history, providing a more flexible environment to summarize and specify the desired editing activities.

Improved collaborative editing environment. With live sharing, different users in different distributed locations may contribute their modeling knowledge and experience seamlessly, such that users in one area can reuse the expertise from those in another, or inexperienced novice users can benefit from the practical experiences of more knowledgeable users.

The more guided editing experience. With live model transformation, users are prompted with guidance about the applicable model transformations during model edit-time, the result being that users can be aware of the available and applicable model transformation patterns and decrease missing reuse opportunities. The specified and summarized model transformation patterns can aid users by facilitating various transformation tasks, such as model creation, error detection and correction [18], aspect-oriented modeling [19], layout configuration [17] and other general model evolution tasks.

On the other hand, although LiveMTBD has the potential to improve reuse and automate the editing activities, several limitations are still present.

The need to ensure the correctness of a live demonstration. Forming the transformation pattern from the editing history is very flexible compared with the explicit demonstration, but it also leads to a possibility that the selected editing operations from the history may not be accurate. For instance, without a mechanism to guide the selection of operations related with certain model elements, extra unnecessary operations could be added accidentally to the pattern, which cannot be filtered by the optimization algorithm; or an incomplete pattern is inferred due to the insufficient operations chosen from the view. Therefore, a crucial aspect for the future work is how to ensure the correctness of the selections from using live demonstration.

Lack of a management feature in the centralized pattern repository. The current implementation of the pattern repository simply stores all the patterns together without classification. This could lead to matching transformation patterns that are not designed for the current modeling language. Therefore, categorizing the patterns is an essential part of the repository management in the future. In addition, the visibility, priority, and authorization of the patterns should also be taken into consideration.

The performance issue of live matching. Matching patterns is expensive in the current implementation of LiveMTBD, which applies a back-tracking algorithm to traverse the selected input candidate pool. This will lead to a performance issue when a large number of transformation patterns are matched by live matching in the editor. Therefore, how to optimize the matching algorithm and improve the performance deserves a deeper investigation.

5 Related Works

The challenges of using MTLs to implement model transformation have been identified previously [10]. Much work has been done to simplify the model transformation implementation processes. Model Transformation By Example (MTBE) was the first attempt in this direction [10]. The idea of MTBE is that instead of writing transformation rules manually, users are asked to build a prototypical set of interrelated mappings between the source and target model instances, and then the metamodel-level transformation rules are semi-automatically generated using a logical programming engine [11]. This approach simplifies model transformation implementation, but is not appropriate for assisting editing activities because: 1) it focuses on direct concept mapping between two different domains rather than changing models within the same domain; 2) they do not support attribute transformation, which is an indispensable editing operation in the modeling process.

Similarly, Brosch et al. introduced a method for specifying composite operations within the user's modeling language and environment of choice [12][13]. The user models the composite operation by-example, changing a source model into the desirable target model. By comparing the source and target states, the specific changes can be summarized by a model difference algorithm, and transformation rules can be generated. This approach focuses on endogenous model transformation, which can be used to assist editing activities. However, attribute transformation has not been considered, and live model transformation and sharing is not currently supported in both these related approaches.

Some works have been done to realize automatic model completion features to create and modify the existing model elements automatically from an incomplete state to a complete state. Sen et al. proposed to transform the metamodel and associated instance models to an Alloy specification, including static semantics [14]. Then, the partial model can be completed automatically by applying a SAT solver. This approach provides guidance to end-users in the model editors, but the limitation is that the inferred complete models are mainly based on the input constraints, rather than enabling end-users to customize their own scenario fully.

Maznek et al. implemented an auto-completion feature for diagram editors [15][16]. Their approach was based on graph grammars. Given an incomplete graph (model) in the editor, all possible graphs that can be generated using the grammar production rules will be suggested to users. Although this is a runtime and live suggestion feature, the suggestions are totally depended on the grammar, the negative consequence is that users need to specify a number to restrict the times of production and avoid infinite loops. Also, the graph grammar may not be fully compatible to process domain-specific modeling languages, and this approach cannot express user-customized editing activities (e.g., the *WECT* must be greater than 300).

General MTLs, particularly graphical MTLs [9] based on left- and right- hand side patterns, can all be extended with a live model transformation feature without much modification, although this is still not a common practice. VIATRA2 [6] already supports

live model transformation matching features. For instance, triggers can be defined as special rules to execute certain model transformations at modeling time. However, a suggestion or guidance before applying the transformation is not available in the editor compared with our approach.

Based on graphical MTLs, Rath et al. performed a detailed investigation on live model transformations using incremental pattern matching techniques [7][8]. They applied the Rete algorithm to preserve the full transformation context in the form of pattern matches that improved the performance of the live transformation. Different from our focus, their live model transformation was mainly aimed at supporting incremental model transformations and model synchronization between source and target models, although it could be applied to automate the editing activities as well. In addition, the full implementation of their approach is based on VIATRA2, which requires the usage of graph transformation rules at the metamodel level. The matching technique they used can be helpful to improve the live matching feature in our approach.

6 Conclusion

In this paper, we presented an extended model transformation environment called LiveMTBD, which supports the specification and reuse of automated model editing activities. Compared with our previous work with MTBD and other by-demonstration and by-example approaches, the main contributions of the current paper are: 1) a demonstration-based transformation inference process that allows a user to identify transformation patterns from the previous edit history using live demonstration; 2) a centralized pattern repository to enable transparent sharing of the transformation patterns; 3) a live model transformation matching engine to provide modeling-time suggestions and user guidance of reusable patterns that match the current modeling context. Our approach is fully implemented and integrated to GEMS.

Acknowledgement

This work is supported by NSF CAREER award CCF-1052616.

References

1. Schmidt, D.: Model-Driven Engineering. *IEEE Computer*, vol. 39, no. 2, pp. 25-32 (2006)
2. Sun, Y., White, J., Gray, J.: Model Transformation by Demonstration. *Model Driven Engineering Languages and Systems (MoDELS)*, Springer-Verlag LNCS 5795, Denver, CO, October 2009, pp. 712-726 (2009)
3. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A Model Transformation Tool. *Science of Computer Programming*, vol. 72, nos. 1/2, pp. 31-39 (2008)

4. Sendall, S., Kozaczynski, W.: Model transformation - The Heart and Soul of Model-Driven Software Development. *IEEE Software*, Special Issue on Model Driven Software Development, vol. 20, no. 5, pp. 42–45 (2003)
5. Gray, J., Lin, Y., Zhang, J.: Automating Change Evolution in Model-Driven Engineering. *IEEE Computer*, Special Issue on Model-Driven Engineering, vol. 39, no. 2, pp. 51–58 (2006)
6. Balogh, Z., Varró D.: Advanced Model Transformation Language Constructs in the VIATRA2 Framework. *Symposium on Applied Computing (SAC)*, Dijon, France, April 2006, pp. 1280–1287 (2006)
7. Rath, I., Bergmann, G., Okros, A., Varro, D.: Live Model Transformations Driven by Incremental Pattern Matching. *International Conference on Model Transformation*, Springer LNCS, vol. 5063, Zurich, Switzerland, pp. 107–121 (2008)
8. Bergmann, G., Rath, I., Varro, D.: Parallelization of Graph Transformation based on Incremental Pattern Matching. *Electronic Communications of EASST 18* (2009)
9. Mens, T., Gorp, P.: A Taxonomy of Model Transformation and its Application to Graph Transformation. *The 1st International Workshop on Graph and Model Transformation, GraMoT'05*, Tallinn, Estonia (2005)
10. Varró D.: Model Transformation by Example. *Model-Driven Engineering Languages and Systems*, Springer-Verlag LNCS 4199, Genova, Italy, October 2006, pp. 410–424 (2006)
11. Balogh, Z., Varró D.: Model Transformation by Example using Inductive Logic Programming. *Software and Systems Modeling*, vol. 8, no. 3, pp. 347–364 (2009)
12. Brosch, P., Langer, P., Seidl, M., Wieland, K., Wimmer, M., Kappel, G., Retschitzegger, W., Schwinger, W.: An Example is Worth a Thousand Words: Composite Operation Modeling By-Example. *International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Spring-Verlag LNCS 5795, Denver, CO, October, 2009, pp. 271–285 (2009)
13. Brosch, P., Seidl, M., Wieland, K., Wimmer, M., Langer, P.: The Operation Recorder: Specifying Model Refactorings By-example. *International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA) – Tool Demonstration*, Orlando, FL, October 2009, pp. 791–792 (2009)
14. Sen, S., Baudry, B., and Vandheluwe, H.: Towards Domain-specific Model Editors with Automatic Model Completion. *SIMULATION*, vol. 86, no. 2, pp. 109–126 (2010)
15. Mazanek, S. and Minas, M.: Business Process Models as a Showcase for Syntax-Based Assistance in Diagram Editors. *International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Spring-Verlag LNCS 5795, Denver, CO, October, 2009, pp. 322–336 (2009)
16. Mazanek, S., Maier, S., and Minas, M.: Auto-completion for Diagram Editors based on Graph Grammars. In *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE Computer Society, Washington, DC, 242–245 (2008)
17. Sun, Y., Gray, J., Langer, P., Wimmer, M., and White, J., “A WYSIWYG Approach for Configuring Model Layout using Model Transformations,” *10th Workshop on Domain-Specific Modeling*, held at SPLASH 2010, Reno, NV (2010)
18. Sun, Y., White, J., Gray, J., Gokhale, A. “Model-Driven Automated Error Recovery in Cloud Computing,” *Model-driven Analysis and Software Development: Architectures and Functions*, IGI Global, Hershey, PA, pp. 136–154 (2009)
19. Sun, Y., Gray, J., Delamare, R., Baudry, B., and White, J.: Automating the Management of Non-functional System Properties using Demonstration-based Model Transformation. *Computer Science - Research and Development*, Springer-Verlag, 2011 (Under review)
20. Generic Eclipse Modeling System (GEMS). <http://www.eclipse.org/gmt/gems/> (2011)
21. Eclipse Epsilon. <http://www.eclipse.org/gmt/epsilon/> (2011)