End-User Support for Debugging Demonstration-Based Model Transformation Execution

Yu Sun[1] and Jeff Gray[2]

[1] University of Alabama at Birmingham, Birmingham AL 35294
yusun@cis.uab.edu
[2] University of Alabama, Tuscaloosa, AL 35401
gray@cs.ua.edu

Abstract. Model Transformation By Demonstration (MTBD) has been developed as an approach that supports model transformation by end-users and domain experts. MTBD infers and generates executable transformation patterns from user demonstrations and refinement from a higher level of abstraction than traditional model transformation languages. However, not every transformation pattern is demonstrated and specified correctly. Similar to writing programs, bugs can also occur during a user demonstration and refinement process, which may transform models into undesired states if left unresolved. This paper presents MTBD Debugger, which is a model transformation debugger based on the MTBD execution engine, enabling users to step through the transformation execution process and track the model's state during a transformation. Sharing the same goal of MTBD, the MTBD Debugger also focuses on end-user participation, so the low-level execution information is hidden during the debugging process.

Keywords: Model Transformation By Demonstration (MTBD), Model Transformation Debug, End-User Programming.

1 Introduction

Model transformation plays an essential role in many applications of Model-Driven Engineering (MDE) [2]. Although a number of model transformation languages (MTLs) have been developed to support various types of model transformation tasks [1], some innovative model transformation approaches and tools have also been introduced to address the complexity of learning and using MTLs, and the challenges of understanding metamodels [16]. Our earlier work on Model Transformation By Demonstration (MTBD) [5], which was influenced by the idea of Model Transformation By Example (MTBE) [3][4][7], enables users to demonstrate how a model transformation should be performed by editing the model instance directly to simulate the model transformation process step-by-step. A recording and inference engine has been developed to capture all user operations and infer a user's intention in a model transformation task. A transformation pattern is generated from the inference, specifying the precondition of the transformation and the sequence of operations

P. Van Gorp, T. Ritter, and L.M. Rose (Eds.): ECMFA 2013, LNCS 7949, pp. 86–100, 2013.

needed to realize the transformation. This pattern can be further refined by users and then executed by automatically matching the precondition in a new model instance and replaying the necessary operations to simulate the model transformation process. This was the focus of our earlier MODELS paper [5].

Using MTBD, users are enabled to specify model transformations without the need to use a MTL. Furthermore, an end-user can describe a desired transformation task without detailed understanding of a specific metamodel. We have applied MTBD to ease the specification of different model transformation activities – model refactoring, model scalability, aspect-oriented modeling, model management and model layout [17][18].

Although the main goal of MTBD is to avoid the steep learning curve and make it end-user centric, there is not a mechanism to check or verify the correctness of the generated transformation patterns. In other words, the correctness of the final transformation pattern totally depends on the demonstration and refinement operations given by the user, and it is impossible to check automatically whether the transformation pattern accurately reflects the user's intention. In practice, this is similar to producing bugs when writing programs. It is also possible that errors will be introduced in the transformation patterns due to the incorrect operations in the demonstration or user refinement step when using MTBD. Incorrect patterns can lead to errors and transform the model into undesired states. For instance, users may perform a demonstration by editing an attribute using the value of a wrong model element; they may give preconditions that are either too restrictive or too weak; or they may forget to mark certain operations as generic (which forces the inferred transformation to be tied to a specific binding).

Obviously, an incorrect transformation pattern can cause the model to be transformed into an incorrect and undesired state or configuration, which may be observed and caught by users. However, knowing the existence of errors and bugs cannot guarantee the correct identification and their location, because MTBD hides all the low-level and metamodel information from users. Also, the final generated pattern is invisible to the end-users, which makes it challenging to map the errors in the target model to the errors in the demonstration or refinement step. This issue becomes even more apparent when reusing an existing transformation pattern generated by a different user, such that the current users who did not create the original pattern have no idea how to locate the source of an error.

In order to enable users to track and ascertain errors in transformation patterns when using MTBD, a transformation pattern execution debugger is needed that can work together with the pattern execution engine. In fact, a number of model transformation debuggers have already been developed for different MTLs [9]. However, the main problem with these debuggers is that they work by tracking the MTL rules or codes, which is at the same level of abstraction as the MTL and therefore not appropriate for some types of end-users and domain experts. Because MTBD has already raised the level of abstraction above the general level of MTLs, the associated MTBD Debugger should be built at the same level of abstraction. Thus, the goal of the MTBD Debugger presented in this paper is to provide users with the

necessary debugging functionality without exposing them to low-level execution details or metamodels.

A brief overview of MTBD will be given in Section 2, followed by an introduction to the MTBD Debugger in Section 3. Section 4 illustrates the usage of the MTBD Debugger for different debugging purposes through several examples. Section 5 summarizes the related work and Section 6 offers concluding remarks.

2 Overview of MTBD

Figure 1 (adapted from [6]) shows the high-level overview of MTBD, which is a complete model transformation framework that allows users to specify a model transformation, as well as to execute the generated transformation pattern in any desired model instances.

The specification of a model transformation using MTBD starts with a demonstration by locating one of the correct places in the model where a transformation is to be made, and directly editing a model instance (e.g., add a new model element or connection, modify the attribute of a model element) to simulate the maintenance task (*User Demonstration*). During the demonstration, users are expected to perform operations not only on model elements and connections, but also on their attributes, so that the attribute composition can be supported. At the same time, an event listener has been developed to monitor all the operations occurring in the model editor and collect the information for each operation in sequence (*Operation Recording*). The list of recorded operations indicates how a non-functional property should be composed in the base model. After the demonstration, the engine optimizes the recorded operations to eliminate any duplicated or meaningless actions (*Operation Optimization*). With an optimized list of recorded operations, the transformation can be inferred by generalizing the behavior in the demonstration (*Pattern Inference*). Because the MTBD approach does not rely on any MTLs, we generate a transformation pattern, which summarizes the precondition of a transformation (i.e., *where* to perform a transformation) and the actions needed in a transformation (i.e., *how* to perform a transformation in this location). Users may also refine the generated transformation pattern by providing more feedback for the precondition of the desired transformation scenario from two perspectives – structure and attributes, or identifying generic operations to be executed repeatedly according to the available model elements and connections.

After the user refinement, the transformation pattern will be finalized and stored in the pattern repository for future use (*Pattern Repository*). The final patterns in the repository can be executed on any model instances. Because a pattern consists of the precondition and the transformation actions, the execution starts with matching the precondition in the new model instance and then carrying out the transformation actions on the matched locations of the model (*Pattern Execution*). The MTBD engine also validates the correctness of the models after each execution process (*Correctness Checking*). Users can choose where to execute the pattern, a sequence of patterns to execute, and the execution times (*Execution Control*). More details about MTBD beyond this summary are in [5].

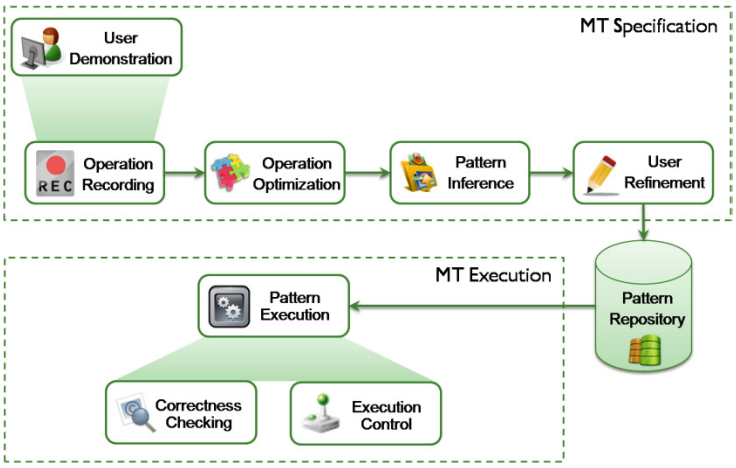

Fig. 1. High-level overview of MTBD (adapted from [6])

3 MTBD Debugger

MTBD Debugger is designed and implemented over the MTBD execution engine. The specific debugging sequence is based on the structure of a transformation pattern. As mentioned in Section 2, a transformation pattern contains the precondition of a transformation (i.e., including the structural precondition and attribute precondition) and the sequence of transformation actions. During the execution of a transformation pattern, any error that is discovered can be traced back to errors in either the precondition or the transformation actions. From the technical perspective as shown in Figure 2, the goal of MTBD Debugger is to help users correctly map the effect of a transformation exerted on the target model instance to the precondition and actions specified in the transformation pattern, so that users can track the cause of an undesired transformation result.

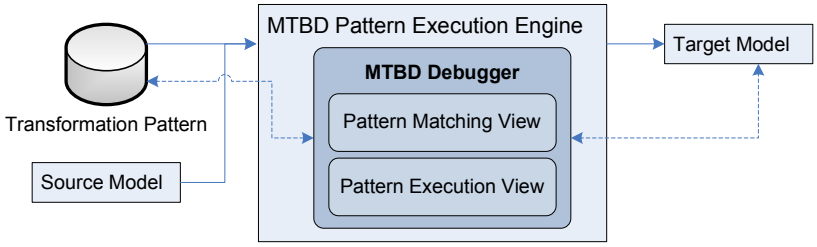

Fig. 2. Overview of MTBD Debugger

The main functionality of the MTBD Debugger is supported by enabling the step through execution of a transformation pattern and displaying the related information with each step in two views – *Pattern Execution View* and *Pattern Matching View*. Users can directly observe what action is about to be executed, what are the matched

model elements for the operation, and more importantly, how the matched elements are determined based on the types of preconditions. This allows the end-users to follow each step and check if it is the desired execution process. In addition, keeping the debugging process at the proper level of abstraction is an essential design decision of MTBD Debugger to assist end-users who are not computer scientists. Similar to MTBD, the MTBD Debugger separates users from knowing any MTLs and hides the low-level execution or metamodel details, so that the same group of users who implement model transformations using MTBD are enabled to debug the same model transformations using the language that represents their domain.

3.1 Pattern Execution View

The *Pattern Execution View* lists all the actions to be executed in a transformation pattern in sequence. As shown in a future example in Figure 5 (which is used later in a specific debugging context), the view displays the type of the action, the main target element used for this action, whether the action is generic or not, and the related details based on the type of the action. In the debugging mode, users can step through each action one-by-one. Before the execution of the action, all the matched elements that will be used for the action are highlighted in the *Pattern Matching View*, so that users can determine which elements will be used for the execution of the action. If the required target element cannot be matched, "null" will be displayed. After the action is executed, the *Pattern Execution View* highlights the next action. At the same time, the model in the editor is updated with the execution of the previous action. Users can check the properties and structure of the latest model instance and determine if it is transformed into the desired state.

3.2 Pattern Matching View

The *Pattern Matching View* works together with the *Pattern Execution View* to provide relevant information about the matched model elements. From Figure 5, it can be seen that it shows the model element type, the precondition associated with it, and the specific model element that is matched in the current model. The list includes all the model elements needed in the transformation pattern. The execution of each action will trigger the highlight of all needed model elements in this view.

4 MTBD Debugger in Action

This section presents a case study that illustrates the use of MTBD Debugger to support tracking and debugging errors in several practical model transformation tasks in a textual game application domain (for the Debugger, we use the same case study from [5] for consistent discussion for those who may refer back to the original MTBD paper).

4.1 Background: MazeGame Case Study

The case study is based on a simple modeling language called MazeGame. A model instance is shown in Figure 3. A *Maze* consists of *Rooms*, which can be connected to each other. Each *Room* can contain either pieces of *Gold*, or a power item such as *Weapon* or *Monster* with an attribute (*strength*) to indicate the power. The goal of the game is to let players move in the *rooms*, collect all pieces of *gold*, and use *weapons* to kill *monsters*. The full Java implementation of the game can be generated automatically from the game configuration specified in the model. We constructed this modeling environment in the GEMS [8] Eclipse modeling tool.

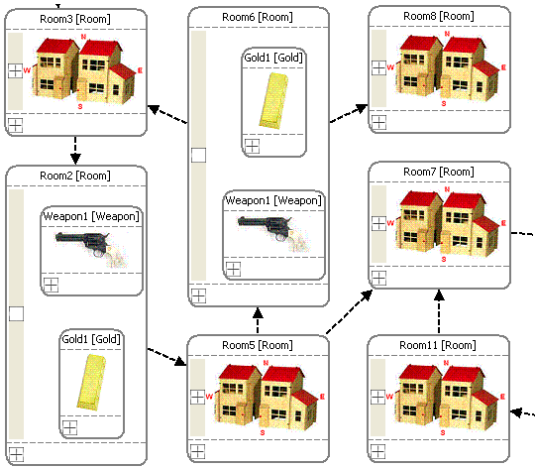

Fig. 3. An excerpt of a MazeGame model instance

Building various game configurations using the MazeGame modeling language often involves performing different model transformation tasks for maintenance purposes. For instance, if there are rooms that contain both *gold* and a *weapon* (the two unfolded rooms in Figure 3, *Room2* and *Room6*), we can implement a model transformation to remove the gold, and replace the *weapon* with a *monster*, with the *strength* of the new *monster* set to half of the *strength* of the replaced *weapon*. Game designers can apply this transformation when they discover that the number of *monsters* is far less than that of *weapons*, making the game too easy (we presented this scenario in [5], but used here for explanation of the MTDB Debugger).

4.2 Debugging in Action

In order to illustrate the usage of MTBD Debugger, we consider transformation errors that end-users may make in this case study when using MTBD, and show how MTBD Debugger can track and locate these errors.

Debugging Example 1. This first example is based on the following transformation task: if a *Monster* is contained in a *Room*, whose *strength* is greater than *100*, replace this *Monster* with a *Weapon* having the same *strength*, and add a *Gold* piece in the same *Room*. Figure 4 shows a concrete example for this transformation task.

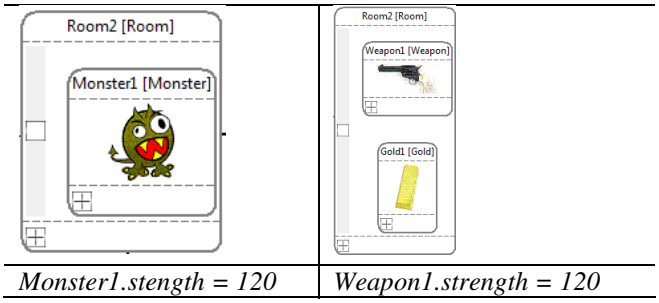

Fig. 4. The excerpt of a MazeGame model before and after replacing the monster

Based on this scenario, a user starts the demonstration by first locating a *Room* with a *Monster* in it, and deleting the *Monster*, followed by adding a *Weapon* plus a *Gold* piece. The *strength* of the new *Weapon* can be configured using the attribute refactoring editor. Finally, a precondition on *Monster* is needed to restrict the transformation (*Monster1.strength* > *100*). As shown in List 1, the user performed all the correct operations except the incorrect precondition was provided (*Monster1.strength* > *10*).

List 1 – Operations for demonstrating replacement of a Monster

Sequence	Operation Performed
1	Remove *Monster1* in *Root.TextGameFolder.Room2*
2	Add a *Weapon* in *Root.TextGameFolder.Room2*
3	Add a *Gold* in *Root.TextGameFolder.Room2*
4	Set *Root.TextGameFolder.Room2.Weapon.strength* = *Monster1.strength* = *120*
5	Set precondition on *Monster1*: *Monster1.strength* > *10*

When applying this generated pattern to the model, it may be found that the transformation takes place in every *Room* with a *Monster* in it even the *strength* of the *Monster* is less than 100, which is not the desired result. Obviously, if the *strength* of every *Monster* is greater than 10, the incorrect precondition can be satisfied with all *Monsters* in the model instance. To debug the error, we execute the transformation pattern again using MTBD Debugger. As shown in Figure 5, the Pattern Execution view lists all the operations to be performed, while the Pattern Matching view provides the currently matched elements for the transformation pattern. Users can step through each of the operations, and the corresponding model elements needed for each operation will be highlighted. For instance, the very first operation in this scenario is to remove the *Monster* in the *Room*. Before executing this operation and stepping to the next one, we can determine which *Monster* is currently matched as the target to be removed. In this case, the *Monster1* in *Room12* is about to be removed. If we check the *strength* attribute of *Monster1* (e.g., *30*), we can observe that there is something wrong with the precondition we specified in the demonstration, because the *strength* of this *Monster* is not greater than *100*. At this point, we can focus on the

precondition in the Pattern Matching view, which shows the actual precondition is "*Strength* > *10*", not "*Strength* > *100*" as desired (the highlighted red box is added to the screenshot to draw attention to the location of the error for readers; this does not appear in the actual tool). The bug is therefore identified and located.

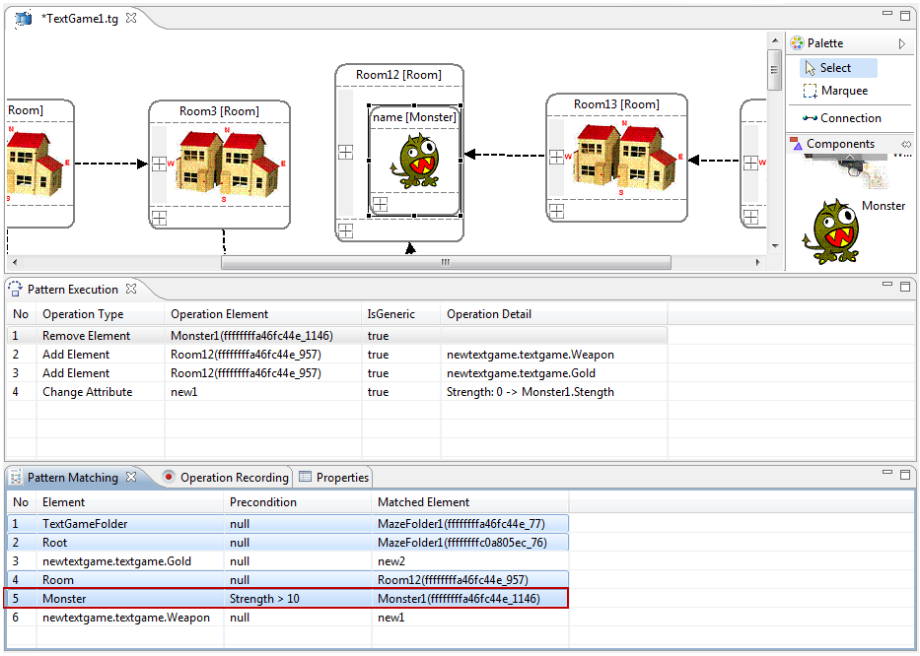

Fig. 5. Debugging the transformation pattern of Example 1

The error in the first example comes from a mistakenly specified precondition that over-matched the model elements. In the second example, we present how to debug a transformation pattern that contains preconditions that are under-matched.

Debugging Example 2. The second example is based on the same transformation scenario as the first one to replace the *Monster* with a *Weapon*. However, in this second demonstration, instead of giving the correct precondition "*Strength* > *100*", the user specified "*Strength* > *1000*" by mistake. As we can imagine, the result of executing this transformation pattern will probably not replace any of the *Monsters* in the model instance, because there are seldom *Monsters* whose *strength* is greater than *1000*.

Similar to the first example, when using the MTBD Debugger to step through the execution process, we can find out the currently matched model elements for each operation. As shown in Figure 6, the first operation to remove the *Monster* contains a null operation element as the target, which means that there is not a *Monster* in the current model instance that can be matched as an operand for this operation. We may think that there is again something wrong with the precondition, so we take a look at the precondition in the Pattern Matching view, and find the precondition set incorrectly as "*Strength* > *1000*".

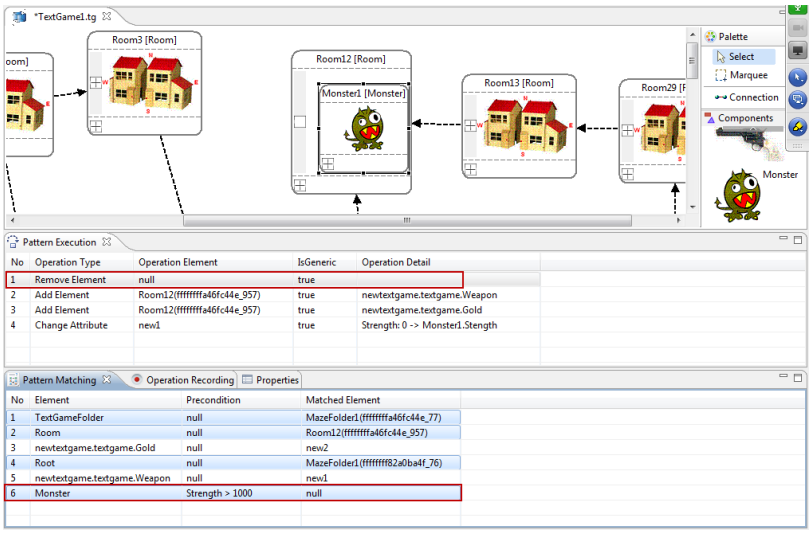

Fig. 6. Debugging the transformation pattern of Example 2

Debugging Example 3. Using MTBD, one of the scenarios that may cause an error is the refinement on the transformation actions in order to identify generic repeatable operations. The third example is based on the scenario that we want to remove all the pieces of *Gold* in all the *Rooms*, no matter how many pieces there are in the *Room* (see Figure 7).

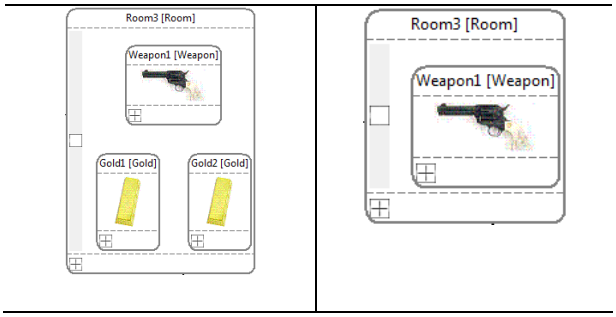

Fig. 7. The excerpt of a MazeGame model before and after removing all Gold

To specify the transformation pattern, a user performs a demonstration on a *Room* that contains two pieces of *Gold* (two operations performed - see List 2).

List 2 – Operations for demonstrating removing all pieces of Gold

Sequence	Operation Performed
1	Remove *Gold1* in *Root.TextGameFolder.Room3*
2	Remove *Gold2* in *Root.TextGameFolder.Room3*

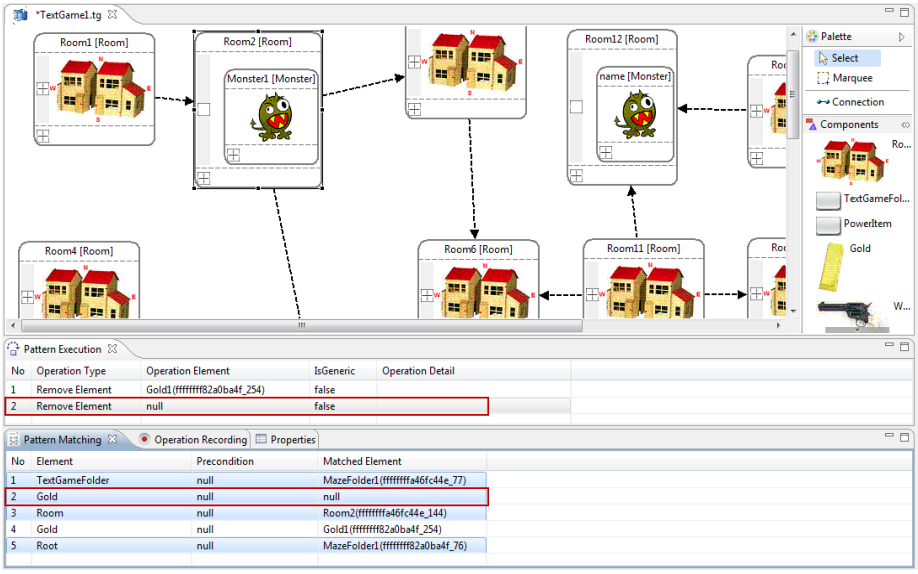

Fig. 8. Debugging the transformation pattern of Example 3

Without giving further refinement on the transformation actions, the user may complete the demonstration. When executing the generated transformation pattern on the model, however, it is found that the Rooms that contain only one piece of *Gold* were not transformed as expected. To track the error, the pattern can be re-executed step-by-step using MTBD Debugger. As listed in the Pattern Execution view, we can see that there are two operations in this pattern, and each operation requires a different target element (i.e., the *Gold* to remove). When the *Room* contains only one piece of *Gold*, the second operation cannot be provided with a correct operand as shown in Figure 8. Thus, the problem of this bug comes from the fact that the transformation actions are not generic so that it always requires a fixed number of model elements to enable the correct transformation. The correct way to use MTBD is to make the demonstration concise, such that users should only demonstrate a single case followed by identifying the necessary generic operations. Thus, the correct demonstration should be done by removing only one piece of *Gold* and then marking it as generic.

Debugging Example 4. Following Example 3, the user may re-demonstrate the removal of *Gold* pieces by only performing a single removal operation. However, the wrong transformation pattern will be generated again due to the user forgetting to mark the operation as generic. This time, when the pattern is executed, only one piece of *Gold* can be removed in each *Room*. To track the error, the MTBD Debugger can reveal whether each operation is generic. When stepping through the execution in *Room3* (Figure 9, which contains two pieces of *Gold*), the user finds that another *Room* is matched after removing only one piece of *Gold*. The user may think that the problem is caused by the generic operations, so by double-checking the generic status, it can be seen from the Pattern Execution view that the removal operation is not generic (the highlighted box marked as false in the middle of the figure).

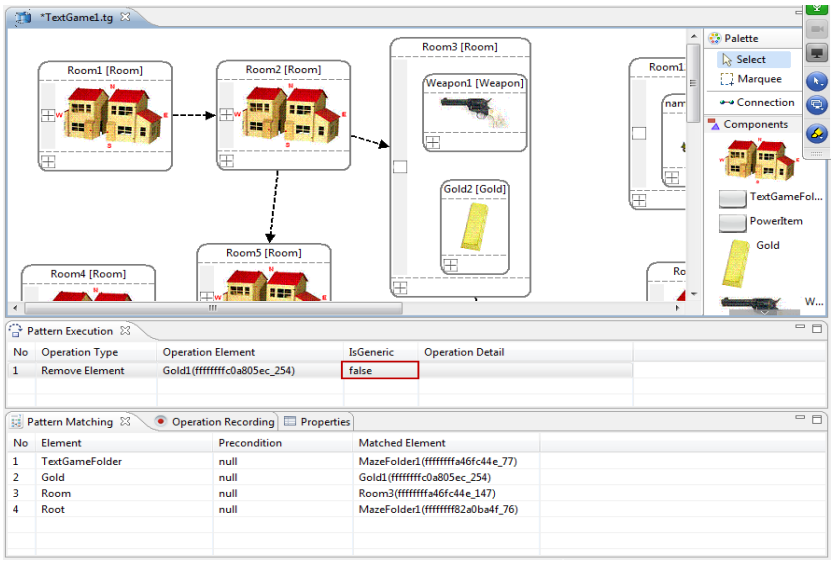

Fig. 9. Debugging the transformation pattern of Example 4

Debugging Example 5. Another common error that occurs when using MTBD is choosing the wrong element in the demonstration process, particularly in the attribute editing demonstration. For example, the user may want to replace all the *Monsters* with *Weapons*, as well as doubling the *strength* of the new *Weapons*, as shown in Figure 10.

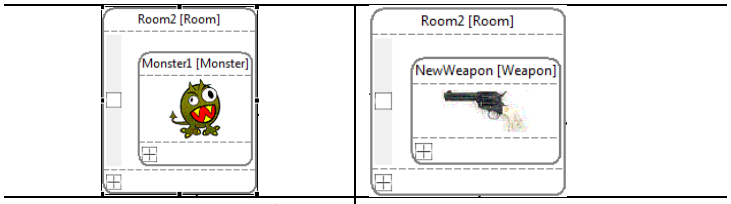

Fig. 10. The excerpt of a MazeGame model before and after doubling the new weapon

The recorded operations are in List 3. An attribute transformation is demonstrated using the attribute refactoring editor. The expected computation of the *strength* is to use the removed *Monster* and double its *strength* value. However, operation 3 in the list mistakenly selects the wrong *Monster* (i.e., *Monster1* in *Room1*) which is not the *Monster* that has just been removed (i.e., *Monster1* in *Room2*). The wrong execution result triggered by this bug is that the new *Weapon* being added in the *Room* uses the strength value of the *Monster* in a different *Room*, which is not what user expects to double.

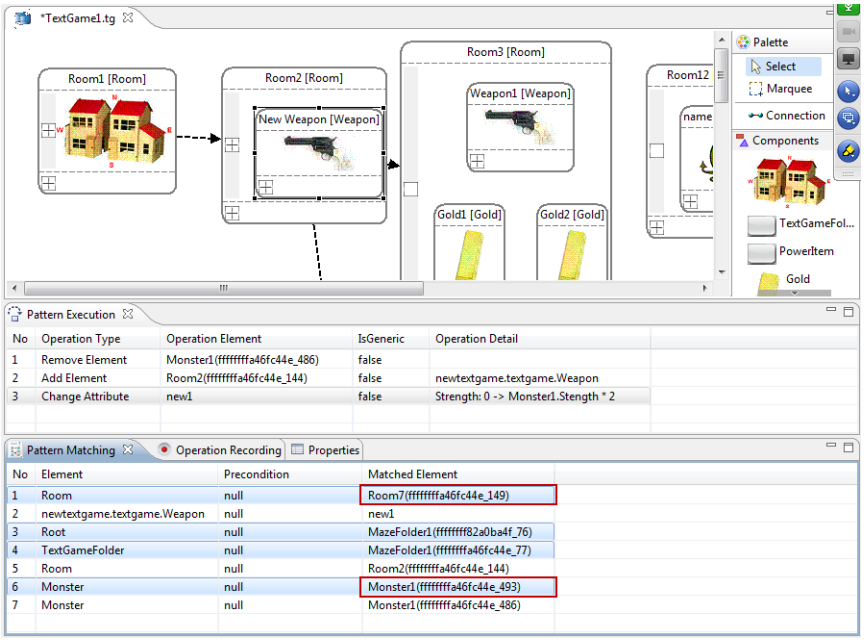

Fig. 11. Debugging the transformation pattern of Example 5

List 3 – Operations for demonstrating replacing a Monster and doubling the strength

Sequence	Operation Performed
1	Remove *Monster1* in *Root.TextGameFolder.Room2*
2	Add a *Weapon* in *Root.TextGameFolder.Room2*
3	Set *Root.TextGameFolder.Room2.Weapon.strength* = *Root.TextGameFolder.Room1.Monster1.strength* * 2 = *152*

This type of bug can be located easily using MTBD Debugger, as shown in Figure 11. When we step through each operation, the used elements in the Pattern Matching view cab be observed. In this case, the remove element operation is done on *Monster1* in *Room2*, while the change attribute operation uses *Monster1* in *Room7*, which means that we probably selected the wrong element in the demonstration of the attribute change process.

5 Related Works

As one of the most popular MTLs, ATL has an associated debugger [9] to provide the basic debugging options similar to general-purpose programming languages, such as step-by-step execution, setting up breakpoints, and watching current variables. Additionally, simple navigation in source and target models is supported. However, all these debugging options are closely related with the language constructs, so it is inappropriate for general end-users who do not have the knowledge of ATL.

Similarly, in the Fujaba modeling environment, Triple Graphical Grammar (TGG) rules [10] can be compiled into Fujaba diagrams implemented in Java, which allows debugging TGG rules directly [11].

Schoenboeck et al. applied a model transformation debugging approach [12] using Transformation Nets (TNs), which is a type of colored Petri Net. The original source and target metamodels are used as the input to derive places in TNs, while model instances are represented as tokens with the places. The actual transformation logic is reflected by the transitions. The derived transformation provides a formalism to describe the runtime semantics and enable the execution of model transformations. An interactive OCL console has been provided to enable users to debug the execution process. TNs are at a higher level of abstraction than MTLs (e.g., QVT is used as the base in this approach), so this approach helps to isolate users from knowing the low-level execution details. Although TNs can be considered as a domain-specific modeling language (DSML) to assist debugging model transformations, it is a different formalism from the specific model transformation area and can be used as a general-purpose specification in many domains, which inevitably limits its end-user friendliness. Most users may find it challenging to switch their model transformation tasks to colored Petri Net transition processes. TNs also aim at defining the underlying operational semantics that are hidden in the model transformation rules, and this exerts an extra burden in its understandability to general end-users and domain experts.

A similar work has been done by Hillberd [13] which presents forensic debugging techniques to model transformation by using the trace information between source and target model instances. The trace information can be used to answer debugging questions in the form of queries that help localize the bugs. In addition, a technique using program slicing to further narrow the area of a potential bug is also shown. Compared with MTBD Debugger, which is a live debugging tool, this work of Hillberd et al. focuses on a different context – forensic debugging. Similar to the ATL debugger, their work aims at providing debugging support to general MTLs used in MDE.

Another related work is focused on debugging a different type of model transformation – Model-to-text (M2T). Dhoolia et al. present an approach for assisting with fault localization in M2T transformations [14]. The basic idea is to create marks in the input-model elements, followed by propagating the marks to the output text during the whole transformation, so that a dynamic process to trace the flow of data from the transform input to the transform output can be realized. Using the generated mark logs and a location where a missing or incorrect string occurs in the output, the user can examine the fault space incrementally.

6 Conclusions and Future Work

Our recent work has focused on tools and concepts that allow end-users to participate in the model transformation process by allowing them to record a desired transformation directly on instance models, rather than applying transformation

languages that may be unfamiliar to them. This paper extends end-user participation in model transformation by presenting a technique that supports end-user debugging of model transformation patterns that were initially recorded through user demonstration. The MTBD Debugger allows users to step through each action in the transformation pattern and check all the relevant information through two views. The MTBD Debugger has been implemented as an extension to the MTBD execution engine and integrated with the MTBD framework.

The MTBD debugger can be applied to the core elements specified in a model transformation pattern. However, one drawback of the current views used in the debugger is that they are textual and not visual. For instance, the Pattern Matching View shows all the needed elements for each action. However, the containment relationship among these elements cannot be seen clearly. It would be very helpful to have another view that shows all the currently involved model elements and their relationships visually. Future work will provide a view that can capture the specific part of the current model that is used for the next transformation action. This can enable users to catch and check the matched elements more easily.

Another option that is useful in the general debugging process, but missing in the MTBD debugger, is the concept of setting a breakpoint. In some large model transformation scenarios (e.g., scaling up a base model to a large and complex state), it is not necessary to watch all the actions being executed one-by-one, so setting a breakpoint would make debugging more useful in this case. Thus, in the Pattern Execution View, it would be helpful to enable the breakpoint setup in the action execution list.

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

References

1. Czarnecki, K., Helsen, S.: Feature-based Survey of Model Transformation Approaches. IBM Systems Journal 45(3), 621–645 (2006)
2. Sendall, S., Kozaczynski, W.: Model Transformation - The Heart and Soul of Model-Driven Software Development. IEEE Software, Special Issue on Model Driven Software Development 20(5), 42–45 (2003)
3. Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards Model Transformation Generation By-Example. In: The 40th Hawaii International Conference on Systems Science, Big Island, HI, p. 285 (January 2007)
4. Varró, D.: Model Transformation By Example. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 410–424. Springer, Heidelberg (2006)
5. Sun, Y., White, J., Gray, J.: Model Transformation By Demonstration. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 712–726. Springer, Heidelberg (2009)
6. Sun, Y.: Model Scalability Using a Model Recording and Inference Engine. In: International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (SPLASH 2010), Reno, NV, pp. 211–212 (October 2010)

7. Balogh, Z., Varró, D.: Model Transformation by Example using Inductive Logic Programming. Software and Systems Modeling 8(3), 347–364 (2009)
8. White, J., Schmidt, D., Mulligan, S.: The Generic Eclipse Modeling System. In: Model-Driven Development Tool Implementer's Forum at the 45th International Conference on Objects, Models, Components and Patterns, Zurich Switzerland (June 2007)
9. Allilaire, F., Bézivin, J., Jouault, F., Kurtev, I.: ATL: Eclipse Support for Model Transformation. In: The Eclipse Technology eXchange Workshop (eTX) of the European Conference on Object-Oriented Programming (ECOOP), Nantes, France (July 2006)
10. Königs, A.: Model Transformation with TGGs. In: Model Transformations in Practice Workshop of MoDELS 2005, Montego Bay, Jamaica (September 2005)
11. Wagner, R.: Developing Model Transformations with Fujaba. International Fujaba Days, Bayreuth, Germany, pp. 79–82 (September 2006)
12. Schoenboeck, J., Kappel, G., Kusel, A., Retschitzegger, W., Schwinger, W., Wimmer, M.: Catch Me If You Can – Debugging Support for Model Transformations. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 5–20. Springer, Heidelberg (2010)
13. Hibberd, M., Lawley, M., Raymond, K.: Forensic Debugging of Model Transformations. In: International Conference on Model Driven Engineering Languages and Systems, Nashville, TN, pp. 589–604 (October 2007)
14. Dhoolia, P., Mani, S., Sinha, V.S., Sinha, S.: Debugging Model-Transformation Failures Using Dynamic Tainting. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 26–51. Springer, Heidelberg (2010)
15. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1/2), 31–39 (2008)
16. Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W., Wimmer, M.: Model Transformation By-Example: A Survey of the First Wave. In: Düsterhöft, A., Klettke, M., Schewe, K.-D. (eds.) Thalheim Festschrift. LNCS, vol. 7260, pp. 197–215. Springer, Heidelberg (2012)
17. Sun, Y., Gray, J., Langer, P., Kappel, G., Wimmer, M., White, J.: A WYSIWYG Approach to Support Layout Configuration in Model Evolution. In: Emerging Technologies for the Evolution and Maintenance of Software Models. Idea Group (2011)
18. Sun, Y., White, J., Gray, J., Gokhale, A.: Model-Driven Automated Error Recovery in Cloud Computing. In: Model-driven Analysis and Software Development: Architectures and Functions, Hershey, PA, pp. 136–155. IGI Global (2009)
19. 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. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 271–285. Springer, Heidelberg (2009)