Model Transformation by Demonstration

Yu Sun[1], Jules White[2], and Jeff Gray[1]

[1] Dept. of Computer and Information Sciences, University of Alabama at Birmingham
{yusun,gray}@cis.uab.edu
[2] Institute for Software Integrated Systems, Vanderbilt University
jules@dre.vanderbilt.edu

Abstract. Model transformations provide a powerful capability to automate model refinements. However, the use of model transformation languages may present challenges to those who are unfamiliar with a specific transformation language. This paper presents an approach called model transformation by demonstration (MTBD), which allows an end-user to demonstrate the exact transformation desired by actually editing a source model and demonstrating the changes that evolve to a target model. An inference engine built into the underlying modeling tool records all editing operations and infers a transformation pattern, which can be reused in other models. The paper motivates the need for the approach and discusses the technical contributions of MTBD. A case study with several sample inferred transformations serves as a concrete example of the benefits of MTBD.

Keywords: Model transformation, Program inference, Refactoring.

1 Introduction

Model transformation is a core part of Domain-Specific Modeling (DSM) and plays an indispensible role in many applications of model engineering (e.g., code generation, model mapping and synchronization, model evolution, and reverse engineering [1]). The traditional way to implement model transformations is to use executable model transformation languages to specify the transformation rules and automate the transformation process [2]. However, the use of model transformation languages may present some challenges to users, particularly to those who are unfamiliar with a specific transformation language. Although declarative expressions are supported in most model transformation languages, the transformation rules are defined at the metamodel level, which requires a clear and deep understanding about the abstract syntax and semantic interrelationships between the source and target models. In some cases, certain domain concepts are hidden in the metamodel and difficult to unveil [3, 4]. These implicit concepts make writing transformation rules challenging. Moreover, a model transformation language may not be at the proper level of abstraction for an end-user and could result in a steep learning curve. One advantage of DSM is that by raising the level of abstraction, domain experts and non-programmers can become participants in software development. However, the difficulty of specifying metamodel-level rules and the associated learning curve may prevent domain experts from

A. Schürr and B. Selic (Eds.): MODELS 2009, LNCS 5795, pp. 712–726, 2009.

contributing to certain model transformation tasks from which they have much domain experience.

Model Transformation By Example (MTBE) is an innovative approach (first introduced in [5]) to address the challenges inherent from using model transformation languages. Instead of writing transformation rules manually, MTBE enables users to define a prototypical set of interrelated mappings between the source and target model instances, and then the metamodel-level transformation rules can be inferred and generated semi-automatically. In this context, users work directly at the model instance level and configure the mappings without knowing any details about the metamodel definition or the hidden concepts. With the semi-automatically generated rules, the simplicity of specifying model transformations is greatly improved.

The current state of MTBE research still has some limitations that may prevent it from being a widely used model transformation approach. The semi-automatic generation often leads to an iterative manual refinement of the generated rules; therefore, the model transformation designers may not be isolated completely from knowing the transformation languages and the metamodel definitions. In addition, the inference of transformation rules depends on the given sets of mapping examples. In order to get a complete and precise inference result, one or more representative examples must be available for users to setup the prototypical mappings, but seeding the process with such examples is not always an easy task in practice. Furthermore, current MTBE approaches focus on mapping the corresponding domain concepts between two different metamodels without handling complex attribute transformations. For instance, in practice, it is quite common to transform an attribute in the source model to another in the target model with some arithmetic or string operations, which is expressed by imperative transformation rules in some transformation languages. Unfortunately, these imperative expressions can only be added manually to the generated rules using current MTBE approaches.

To further simplify the model transformation process, we propose a new approach – Model Transformation By Demonstration (MTBD). Instead of the MTBE idea of inferring the rules from a prototypical set of mappings, users are asked to demonstrate how the model transformation should be done by directly editing (e.g., add, delete, connect, update) the model instance to simulate the model transformation process step by step. A recording and inference engine has been developed, as part of a prototype called MT-Scribe, 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 needed to realize the transformation. This pattern can be reused by automatically matching the precondition in a new model instance and replaying the necessary operations to simulate the model transformation process.

We have successfully applied this approach to implement endogenous model transformations, where both the source and target models conform to the same metamodel. Our initial experience in using MTBD suggests improvement in the efficiency and simplicity of specifying model transformations. The current contributions of MTBD include the following:

- MTBD represents one of the first attempts to simplify the specification of endogenous model transformations (in contrast to the exogenous focus of

previous MTBE approaches), which offers improvement for automating model evolution activities (e.g., model refactoring, scaling, and aspect weaving).
- MTBD can be used to specify model transformations without the need to use a model transformation language. Furthermore, an end-user can describe a desired transformation task without detailed understanding of a specific metamodel.
- The current status of MT-Scribe includes: (1) a recording engine to completely capture all user operations and related context; (2) an algorithm to optimize the recorded operations, eliminating meaningless operations; (3) an algorithm to automatically match a transformation precondition in any model instance; (4) support to infer transformations with attribute operations; (5) a correctness checking and undo mechanism to guarantee the correctness of the transformation process; (6) fully automatic generation of a transformation pattern, without iterative manual refinement.

The rest of this paper is organized as follows. A motivating example is first given in Section 2. The paper demonstrates the concept of MTBD through two endogenous model transformation examples. Section 3 presents the overview and main steps of our approach, followed by an explanation of the technical implementation and algorithms through a running example in Section 4. An additional example is also given at the end of Section 4 to further illustrate the idea. Related transformation techniques are compared in Section 5, and Section 6 offers concluding remarks and summarizes future work.

2 MazeGame – A Motivating Example

This section introduces an endogenous transformation task in a simple modeling language called MazeGame. For the purpose of introducing MTBD, the MazeGame examples presented in this paper are simple transformation cases in a small domain. From the metamodel definition in Figure 1, a maze consists of rooms, which can be connected to each other. Each room can contain gold, a weapon or a monster with the `powerValue` attribute to specify the power. This modeling language is used to generate a textual game in Java, enabling players to type textual commands to move in the maze and collect all the gold without being killed by monsters. A model instance describes a specific maze configuration. Collecting weapons during game-play increases a player's power, which can be used to kill monsters. We constructed this metamodel in GEMS (Generic Eclipse Modeling System) [6]. A model instance is shown in Figure 2.

In the context of this domain, a transformation task can be specified as: for those rooms that contain gold and a weapon (the two unfolded rooms in Figure 2, Room2 and Room6), the transformation removes one gold piece, replaces the weapon with a monster, and sets the `powerValue` of the new monster to be half of the `powerValue` of the weapon being replaced. This transformation is used when the maze designer discovers that the number of monsters is far less than that of weapons, making the game too easy.

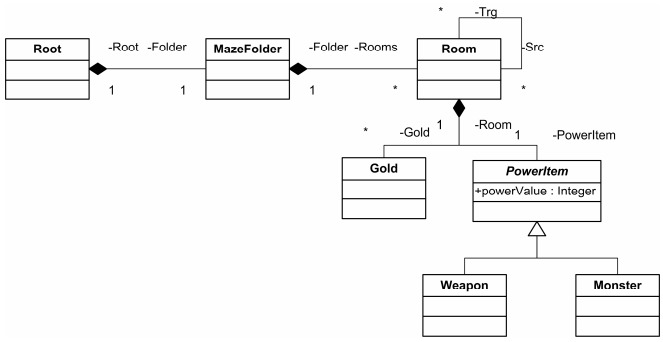

Fig. 1. The MazeGame metamodel

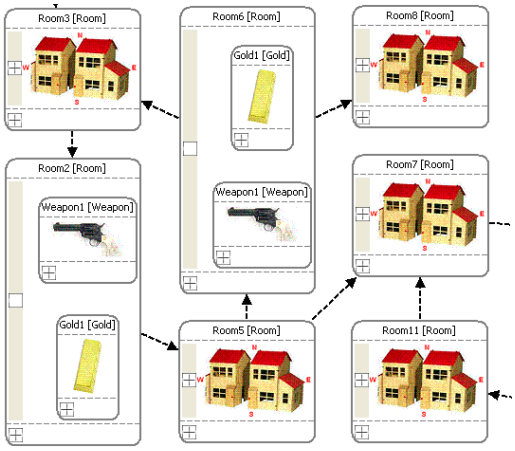

Fig. 2. Part of a MazeGame model instance

Some model transformation languages supporting endogenous transformation (e.g., ATL [7] and C-SAW [8]) can be used to complete this task by specifying the transformation rules. However, domain experts, or in this case, maze designers who have very little knowledge about computer science may find it challenging to learn a transformation language and understand the metamodel definition. To use MTBE, the appropriate source and target models are needed that fit the desired transformation task. Such examples may not be readily available and may require a large amount of time to create for large models. Also, the attribute modify operation (e.g., transforming the `powerValue` of the weapon) cannot be inferred and generated automatically by existing MTBE approaches.

3 Overview of MTBD

MTBD is motivated by the difficulties of learning new model transformation languages and understanding metamodel definitions, and the limitations of MTBE. By analyzing the recorded user operations, a transformation pattern can be inferred and

then reused by automatic pattern matching without the availability of model transformation language support in a modeling tool. The MTBD process (Figure 3) consists of five main steps.

Step 1: User demonstration and operations recording. A user-recorded demonstration provides the base for transformation pattern analysis and inference, so accurately recording all user operations is the first step. The demonstration is given by directly editing a model instance (e.g., add a new model element or connection, modify the attribute of a model element) to simulate a transformation task. An event listener has been developed as part of MT-Scribe to monitor all the operations occurring in the model editor. For each operation that is captured, all the information about the operation is encapsulated into an object, similar to a Command pattern. Finally, the list of objects represents the sequence of operations needed to finish a transformation task.

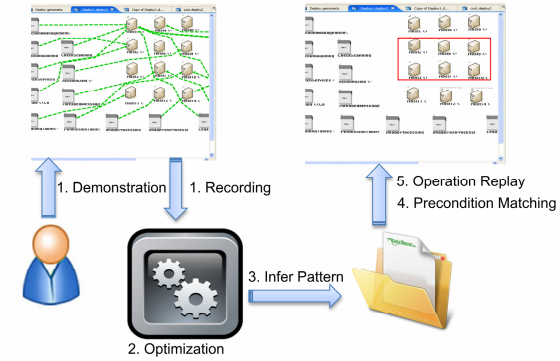

Fig. 3. MTBD overview

Step 2: Optimize recorded operations. The sequence of operations recorded directs how a transformation should be performed. However, not all operations are meaningful. For instance, without a careful design of the demonstration, it is possible that a user first adds a new element and modifies its attributes, and then deletes it in another operation; the result being that all the operations regarding this element actually did not take effect in the transformation process and therefore are meaningless. The presence of meaningless operations not only has the potential to make the inferred transformation preconditions inaccurate, but also exerts a negative influence on the efficiency of a transformation, especially when it executes on a large model instance. Thus, an optimization that eliminates all meaningless operations is automatically done after the recording.

Step 3: Infer the transformation pattern. Because our approach does not rely on a model transformation language, it is not necessary to generate specific transformation rules; instead, a general transformation pattern is inferred. This pattern describes the precondition of a transformation (i.e., where the transformation should be performed) and the actions of a transformation (i.e., how the transformation should be realized). By analyzing the recorded operations, the related meta-information of model elements and connections is extracted to construct the precondition, while the actions are specified by the operation sequence.

Step 4: Precondition matching. After a pattern is summarized, it can be reused and applied to any model instance from the same metamodel. By selecting a pattern from the repository, the MT-Scribe engine automatically traverses the model instance to search all locations that match the selected pattern. A notification is given if no matching locations are found. In MTBD, a matching location contains the necessary model elements and connections on which the recorded operations could be executed correctly.

Step 5: Replay operations and correctness checking. When a matching location is found, the recorded operations are replayed to transform the current model instance. The pattern matching step guarantees that operations can be executed with necessary operands. However, it does not ensure that executing them will not violate the metamodel. Therefore, each applied operation is logged and model instance correctness checking is performed after every operation execution. If a certain operation violates the metamodel definition, all executed operations are undone and the whole transformation replay is cancelled.

4 Technical Implementation and Algorithms Supporting MTBD

An Eclipse plug-in has been implemented in GEMS to realize the MTBD approach. To illustrate the usage and implementation of each step from Section 3, the process of inferring a user-demonstrated transformation is presented using the motivating example. A second MazeGame transformation is introduced to illustrate the idea further.

4.1 Demonstration of MTBD Using the MazeGame

GEMS provides an extension point to capture all events that occur during user interaction on a model instance. To infer a transformation pattern, the model editing operations performed by the user must be recorded. In GEMS, user operations can be classified into six categories. By filtering out unrelated events, all operations are recorded in sequence and stored as operation objects, with the necessary information encapsulated as listed in Table 1. The final list of operation objects serves as the fundamental knowledge base for the pattern inference and summary in later steps.

Table 1. Six types of recorded user operations

Operation Type	*Information Recorded*
Add an Element	Location of the parent element and its meta type The newly added element and its meta type
Remove an Element	Location of the element being removed and its meta type
Modify an Element	Location of the element being modified and its meta type The attribute name, the old value and the new value
Add a Connection	Location of the parent source and target elements and their meta types The newly added connection and its meta type
Remove a Connection	Location of the connection being modified and its meta type
Modify a Connection	Location of the connection being modified and its meta type The attribute name, the old value and the new value

To demonstrate the transformation in the motivating example from Section 2, a user must first find a room that contains a gold piece and a weapon. The four operations listed in Table 2 are performed during the user demonstration. The whole model changing process is shown in Figure 4. The remove and add operations in the first three steps are realized by the basic editing actions in the editor, and the fourth operation to modify the attribute is implemented by choosing the attribute from the attribute tree and specifying the arithmetic expression. When the `powerValue` of `weapon1` is chosen, its value (80) is displayed. A user may type " / 2" in the expression editor to identify the way that a numeric attribute is changed by a transformation. By clicking the evaluate button in the recording dialog, the final value for the attribute (40) is calculated and assigned to the current attribute being edited. As a result, the recording has defined this attribute operation as "`monster1.powerValue = weapon1.powerValue / 2`". In this way, constant values and formulas are typed directly, while referenced attributes are selected from the attribute tree. Attribute computation is therefore enabled in the user demonstration process using a real model instance, rather than at the metamodel level. This is currently impossible in most MTBE implementations.

Table 2. The sequence of operations demonstrated to realize motivating example

No.	*Operation*	*Information Recorded*
1	Remove Gold1	Location: Root1.MazeFolder1.Room2.Gold1 Meta type: Root.MazeFoler.Room.Gold
2	Remove Weapon1	Location: Root1.MazeFolder1.Room2.Weapon1 Meta type: Root.MazeFoler.Room.Weapon
3	Add a Monster	Location: Root1.MazeFolder1.Room2 Meta type: Root.MazeFoler.Room New element: Monster1 Meta Type: Monster
4	Modify Monster1	Location: Root1.MazeFolder1.Room2.Monster1 Meta type: Root.MazeFolder.Room.Monster Attribute: powerValue Old value: 0 New value: Weapon1 / 2

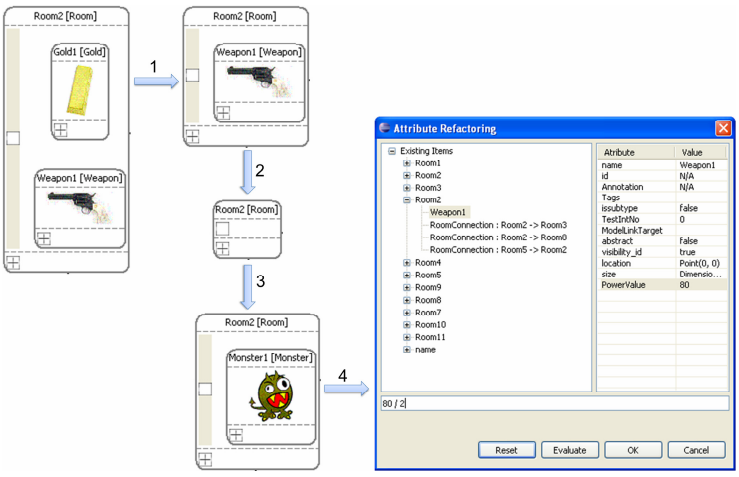

Fig. 4. Model changing process

Algorithm 1. Optimize Operation List

```
for each op in the input operation list
  switch (op.type)
    case ADD_ELEM:
      for each op_temp after the current op in the list
        if op_temp.type == REMOVE_ELEM and op_temp removes what op added
          then remove both op and op_temp from the list
      end for
    case MODIFY_ELEM:
      traverse the final model instance and search the element being modified
      if not found then remove op from the list
      if found then compare the attribute value with the value stored in op
        if different then remove op from the list
    case ADD_CONN:
      for each op_temp after the current op in the list
        if op_temp.type == REMOVE_CONN and op_temp removes what op added
          then remove both op and op_temp from the list
      end for
    case MODIFY_CONN:
      traverse the final model instance and search the connection being modified
      if not found then remove op from opList
      if found then compare its attribute value with the value stored in op
        if different then remove op from opList
end for
```

Given the final list of recorded operations and the final model instance after user demonstration, an optimization phase that removes meaningless operations is performed by analyzing each operation in the list of recorded operations. The optimization algorithm is given in Algorithm 1. Based on the optimized operation list, the transformation pattern is inferred. Because no transformation language is used in the inference, the result is called a transformation pattern rather than a transformation rule (Please note: we can also generate concrete transformation rules from the inferred pattern). A transformation pattern consists of a precondition and the transformation actions.

Table 3. Model object list in precondition

elem1.elem2.elem3.elem4
elem1.elem2.elem3.elem5
elem1.elem2.elem3 (elem6)
elem1.elem2.elem3.elem6

Table 4. Model objects type table

Model Object	Meta Type
elem1	Root
elem2	MazeFolder
elem3	Room
elem4	Gold
elem5	Weapon
elem6	Monster

Table 3 and Table 4 together specify the precondition of the example inferred from the operation list, i.e., all the rooms that contain a gold piece and a weapon. The inference is accomplished by extracting the meta information of the recorded operations and generalizing them. In Table 3, elem6 in parenthesis denotes a newly added

element. Instead of the specific IDs in the recorded operations, generic names with a meta type mapping table are used to describe the precondition. This precondition guarantees that the operations could be executed correctly with sufficient operands. To implement more powerful transformations, more complex preconditions need to be enabled, which are mentioned in Section 6. Table 5 gives the actions as recorded operations with generic element names. The summarized transformation pattern is serialized and stored in a pattern repository.

Table 5. Transformation actions with generic element names

Remove elem4 from elem3
Remove elem5 from elem3
Add elem6 in elem3
Modify elem6: elem6.powerValue = elem5.powerValue / 2

To apply a reusable transformation to a model instance, a pattern is selected from the repository and applied to a portion of the model instance. The MTBD plug-in will traverse the model instance and find all locations that match the precondition in the pattern. The backtracking algorithm (Algorithm 2) is used in the matching process. To enable more flexible matching in the model instance, two matching modes are supported. The default mode traverses the whole model instance to search all locations that match the precondition. A customized mode assists users in selecting parts of the model instance to traverse. In either mode, the MTBD plug-in reports if no locations are matched on a specific model instance.

After a precondition is matched to a location in the model instance, the transformation operations will be replayed automatically in sequence to realize the transformation process. Because an operation is implemented by low-level APIs provided by GEMS, an operation might be executed without consideration by the model correctness checking mechanism in the model editor. The possible result is a metamodel violation (e.g., if an operation is to add a monster in a room, it can still be replayed in a room even if a monster already exists, but at most one monster is allowed to be in a room according to the metamodel). To ensure a correct transformation, the model correctness checking is done after replaying each operation by calling the GEMS model checking module. Each replay is also logged in a stack, so that if a violation occurs, the replayed operations can be undone and rolled back to restore the original model instance.

Algorithm 2. Precondition Matching

```
initialize a candidate object list of all the elements and connections in the selected model instance
for each entry e in the model object list
  for each obj in the candidate object list
    if obj matches e then assign obj to e and break
    if obj does not match e then continue
  end for
  if e is assigned and is the last entry in the list then matching succeeds
  if e has not been assign then backtrack the previous e and try again
  if no further backtracking is allowed then matching fails
end for
```

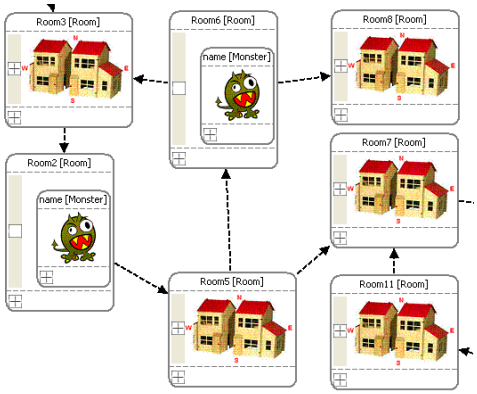

Fig. 5. Part of the model instance after the transformation

Figure 5 shows part of the model instance after applying the transformation pattern example in a MazeGame model instance. The gold piece is removed and the weapon is replaced with a monster whose `powerValue` is half as before. The whole process with more detailed information can be viewed as a video on the project website [22].

4.2 Another Transformation Example: Balancing Game Play

A perfect maze game has a balance of power between monsters and weapons, so winning a game should be neither too easy nor too hard. To make the motivating example more difficult to play, a weapon is replaced with a monster if it is in a room together with a gold piece, so that getting a piece of gold first requires killing a monster. In some other cases, the game may accidentally become too difficult. For example, sometimes monsters appear in a sequence of rooms and the consequence is that a player will encounter several monsters in a row. Figure 6 (left) shows a part of a maze where three rooms connected with each other all contain a monster. To balance the power, it is necessary to replace the monster in the third room with a weapon, and the `powerValue` of the weapon defined as the sum of the monsters in the first two rooms. In addition, to avoid encountering three monsters in a row, it is necessary to reconnect the rooms so that the first monster room connects to the third weapon room, which then connects to the second monster room, as shown in Figure 6 (right). Table 6 indicates the operations needed for this transformation.

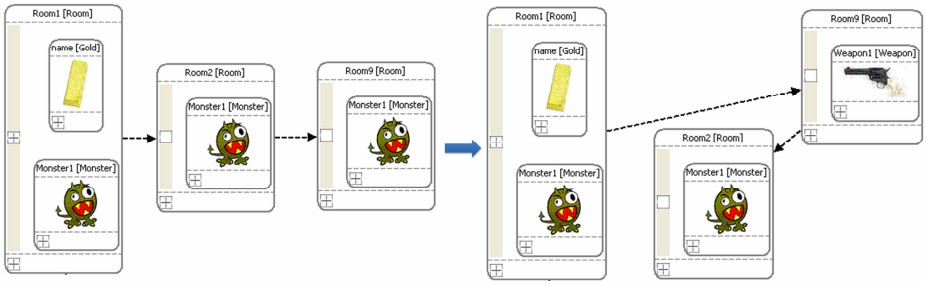

Fig. 6. Avoid encountering three monsters in a row

Table 6. The sequence of operations demonstrated to avoid three monsters in a row

No.	*Operation*
1	Remove Monster1 in Room 9
2	Add a new Weapon in Room 9
3	Set the powerValue of the new weapon to be the sum of two monsters in Room1 and Room2
4	Remove the connection from Room1 to Room2
5	Remove the connection from Room2 to Room9
6	Add a connection from Room1 to Room9
7	Add a connection from Room9 to Room2

The final generated transformation precondition is shown in Tables 7 and 8, representing all three rooms connected one-by-one with each containing a monster. The stored transformation actions are listed in Table 9. Even though this is a very simple case study, the transformation of `powerValue` in this example transformation cannot be accomplished by MTBE, which mainly focuses on direct concept mappings. Our MTBD approach provides an opportunity to define a computation used within a transformation.

Table 7. Model object list in precondition

elem1.elem2.elem3.elem4
elem1.elem2.elem3 (elem5)
elem1.elem2.elem6.elem8
elem1.elem2.elem7.elem9
elem1.elem2.conn1:elem6->elem7
elem1.elem2.conn2:elem7->elem3
elem1.elem2(conn3:elem6->elem3)
elem1.elem2(conn4:elem3->elem7)

Table 8. Model objects type table

Model Object	Meta Type
elem1	Root
elem2	MazeFolder
elem3	Room
elem4	Monster
elem5	Weapon
elem6	Room
elem7	Room
elem8	Monster
elem9	Monster
conn1	RoomConnection
conn2	RoomConnection
conn3	RoomConnection
conn4	RoomConnection

Table 9. Transformation actions with generic element names

Remove elem4 from elem3
Add elem5 in elem3
Modify elem5: elem5.powerValue = elem8.powerValue + elem9.powerValue
Remove conn1
Remove conn2
Add conn3 from elem6 to elem3
Add conn4 from elem3 to elem7

5 Related Work in Model Transformation Inference

MTBD aims to simplify implementation of model transformation tasks, following the similar direction of MTBE approaches. Balogh and Varró introduced MTBE by using inductive logic programming [9, 10]. The idea is to generate graph transformation rules from a set of user-defined mappings between the source and target model instances by applying an inductive logic engine. Similarly, Strommer and Wimmer implemented an Eclipse prototype to enable generation of ATL rules from the semantic mappings between domain models [11, 12]. Both approaches provide semi-automatic generation of model transformation rules, which need further refinement by a user. Because both approaches are based on semantic mappings, they are more appropriate in the context of exogenous model transformations between two different metamodels. However, the generation of rules to transform attributes is not well supported in most MTBE implementations.

MTBD and MTBE are actually extensions of the "by-example" concept. Query-by-example [13] provides a graphical query interface to enable users to use visual tables to specify example query elements and conditions. A similar idea to our approach is called programming-by-example [14, 15], which is used to infer new behaviors by demonstrating the actions on concrete examples. In addition, the "by-example" idea has also been applied to XML document transformation [16]. XML schema transformers can be derived from examples, which then generate XSLT code to transform XML documents.

Although our contribution focuses on model transformations, a similar work has been done to carry out program transformations by demonstration [17]. To perform a program transformation, users first manually change a concrete program example, and all the changes will be recorded by the monitoring plug-in. Then, the recorded changes will be generalized in a transformation. After editing and specifying the generated transformation, it can be applied to other source code locations. Although it also supports the specification of how variable values are computed, it is in a separate step with much manual editing involved. MTBD automates this step in the demonstration process and is focused on demonstrating changes on model instances, not source code.

6 Conclusion and Future Work

This paper introduces a new approach to simplify model transformation tasks, which does not rely on any model transformation language or the understanding of a specific metamodel. To avoid iterative user refinements after the generation process, we made the process fully automated by enabling users to demonstrate not only the transformation precondition, but also the transformation actions, including attribute computations. The generated transformation patterns are stored in a repository, which can be applied to any model instance in the same metamodel from which the transformation was recorded. A complete Eclipse plug-in for GEMS, called MT-Scribe, has been developed to implement the MTDB approach. The examples presented in this paper

are simple transformation cases in a small domain, which are used to focus on the MTBD approach. The examples also illustrate the type of challenges that are encountered when using the direct mapping approach of most MTBE implementations. The current implementation can also carry out other complex transformations in practical domains, such as UML refactoring. More details and examples about MTBD, including video demonstrations, are available at [22]. The MTBD idea can be applied to improve many model evolution tasks, such as:

- **Model refactoring.** Like program refactoring, model refactoring improves the internal structure of a system model without changing its external behavior. The traditional approach is to use a model transformation engine with a refactoring language [18]. The current version of MTBD can be used to support model refactoring. MTBD allows end-users to build a set of reusable refactorings that are domain-specific.
- **Aspect-Oriented Modeling (AOM).** AOM addresses crosscutting concerns in models by separating each concern and weaving it within a base model. Aspects can be defined by using either a textual constraint language [19] or graphical modeling language [20]. MTBD can also be applied to automate AOM, with preconditions representing pointcuts and the transformation actions corresponding to advice.
- **Model scalability.** A model transformation engine can be used to scale model instances, such as the replicators implemented in [21]. Instead of specifying scaling rules, users can demonstrate the scaling process by using MTBD.

We believe that MTBD can also be used in exogenous model transformations. In a modeling environment where editing two model instances from two different domains are allowed, users can edit the source model and change it to the desirable target model. Then the transformation pattern or rules could be inferred from the editing operations, which is our future main focus.

In the current version of MT-Scribe, one limitation is that only the basic or the weakest precondition can be inferred. For instance, in the motivating example, the precondition is all the rooms that contain at least one gold piece and one weapon. However, it is impossible to further restrict it to only the rooms with more than two connections and the `powerValue` of the contained weapon is more than 100. This inflexibility of specifying preconditions exerts a negative influence on the power of MTBD. To enable more powerful precondition definitions, we will implement one more step to ask users to demonstrate the precondition as well. Users will be asked to select the model objects in the editor and setup the conditions that need to be satisfied.

Furthermore, the attribute operations currently supported are only basic arithmetic operations and string concatenation. However, more powerful operations and functions (e.g., `max()` and `min()`) are available in some model transformation languages. Hence, to make MTBD more practical, these additional attribute operations should be supported in the demonstration process.

Acknowledgement. This work was supported by NSF CAREER award CCF-0643725.

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: Proceedings of the 40th Hawaii International Conference on Systems Science, Big Island, HI, January 2007, p. 285 (2007)
4. Kappel, G., Kapsammer, E., Kargl, H., Kramler, G., Reiter, T., Retschitzegger, W., Schwinger, W., Wimmer, M.: Lifting metamodels to ontologies - a step to the semantic integration of modeling languages. In: Proceedings of International Conference on Model Driven Engineering Languages and Systems, Genova, Italy, October 2006, pp. 528–542 (2006)
5. Varró, D.: Model transformation by example. In: Proceedings of Model Driven Engineering Languages and Systems, Genova, Italy, October 2006, pp. 410–424 (2006)
6. Generic Eclipse Modeling System (GEMS), `http://www.eclipse.org/gmt/gems/`
7. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1/2), 31–39 (2008)
8. Gray, J., Lin, Y., Zhang, J.: Automating change evolution in model-driven engineering. IEEE Computer, Special Issue on Model-Driven Engineering 39(2), 51–58 (2006)
9. Balogh, Z., Varró, D.: Model transformation by example using inductive logic programming. In: Software and Systems Modeling. Springer, Heidelberg (2009)
10. Varró, D., Balogh, Z.: Automating model transformation by example using inductive logic programming. In: Proceedings of the 2007 ACM Symposium on Applied Computing, Seoul, Korea, March 2007, pp. 978–984 (2007)
11. Strommer, M., Wimmer, M.: A framework for model transformation by-example: Concepts and tool support. In: Proceedings of the 46th International Conference on Technology of Object-Oriented Languages and Systems, Zurich, Switzerland, July 2008, pp. 372–391 (2008)
12. Strommer, M., Murzek, M., Wimmer, M.: Applying model transformation by-example on business process modeling languages. In: Proceedings of Third International Workshop on Foundations and Practices of UML, Auckland, New Zealand, November 2007, pp. 116–125 (2007)
13. Zloof, M.: Query-By-Example: The invocation and definition of tables and terms. In: Proceedings of International Conference on Very Large Data Bases, Framingham, Massachusetts, 1975, pp. 1–24 (1975)
14. Cypher, A. (ed.): Watch what I do: Programming by demonstration. MIT Press, Cambridge (1993)
15. Lieberman, H.: Special issue on Programming by example. Communication of ACM 43(3), 72–114 (2000)
16. Lechner, S., Schrefl, M.: Defining web schema transformers by example. In: Mařík, V., Štěpánková, O., Retschitzegger, W. (eds.) DEXA 2003. LNCS, vol. 2736, pp. 46–56. Springer, Heidelberg (2003)
17. Robbes, R., Lanza, M.: Example-based program transformation. In: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems, Toulouse, France, October 2008, pp. 174–188 (2008)

18. Zhang, J., Lin, Y., Gray, J.: Generic and domain-specific model refactoring using a model transformation engine. In: Model-driven Software Development, ch. 9, pp. 199–218. Springer, Heidelberg (2005)
19. Zhang, J., Cottenier, T., Berg, A., Gray, J.: Aspect composition in the Motorola aspect-oriented modeling weaver. Journal of Object Technology, Special Issue on Aspect-Oriented Modeling 6(7), 89–108 (2007)
20. Balasubramanian, K., Gokhale, A., Lin, Y., Zhang, J., Gray, J.: Weaving deployment aspects into domain-specific models. International Journal on Software Engineering and Knowledge Engineering, Special Issue on Aspect-Oriented Modeling 16(3), 403–424 (2006)
21. Gray, J., Lin, Y., Zhang, J., Nordstrom, S., Gokhale, A., Neema, S., Gokhale, S.: Replicators: Transformations to address model scalability. In: Proceedings of Model Driven Engineering Languages and Systems, Montego Bay, Jamaica, October 2005, pp. 295–308 (2005)
22. MTBD Project Page, `http://www.cis.uab.edu/softcom/mtbd`