Incremental Model Synchronization by Bi-Directional Model Transformations

István Madari, László Lengyel, Gergely Mezei
Budapest University of Technology and Economics
Department of Automation and Applied Informatics
{istvan.madari, lengyel, gmezei}@aut.bme.hu

Abstract—Model transformation is a focused area in Model-Driven Software Development. With the help of model transformations we can generate source code, other models and documentation from the source models. During the development, a recurring problem is that the source and target artifacts coexist and they evolve independently. This means that after the transformation the target artifacts can be changed by the developer. The problem in this case is that the target artifact will not be consistent with the source model. One option to maintain consistency is by synchronizing our artifacts with model transformation. With the help of synchronization, the developer can work on each artifact, because they are consistent. However the synchronization can be quite complex and cannot be applied in many cases. Usually the inverse transformation does not exist, or it cannot be determined uniquely. This paper presents how we can track the modifications of the transformation, and how we can use this information in the synchronization process.

I. INTRODUCTION

In model-based software engineering, developers use numerous models to define a system. From these models we can generate source code, as result of which the artifacts in the code space and the model space coexist. This means that whenever the developer changes the generated artifact, it will not be consistent with the input model. In model-based software engineering the generation is often implemented by model transformations: if the developer modifies the target, the model transformations offer a potent way to trace the changes back to the source. After the synchronization the developer can change the synchronized input model, and generate the source code again. We have several models and source code, which must be consistent with each other. The aim of the synchronization process is to keep up this consistency without breaking the existing work.

We cannot express everything in one model. We can develop the user interface with modeling systems much easier than manual coding. However, the behavior of the application cannot be modeled in detail, in which case it is easier to write source code manually. This means that if the target artifacts are generated, and are subsequently modified by the developer, the changes cannot be traced back, because the source model cannot express these modifications. Notice that we used source model only to express the user interface, not the behavior. During the synchronization process we have to keep this information. In order to do so we want to use incremental model synchronization between the source and the target. This means that the synchronization process does not generate the whole artifacts from the beginning. Figure 1 depicts the procedure of the development, using incremental synchronization.

![Fig. 1. (a) Edit the source model; (b) Generate the target artifact; (c) Modify the target; (d) Trace back the changes](image)

This is the simpler case: the source and the target models are not edited simultaneously. The synchronization process is more complicated if both the target and source models are changed.

One of the current research areas is developing transformations that can realize the synchronization process between models. Several possibilities exist to develop model transformations for sake of synchronization. We can either develop two unidirectional transformations between the source and the target, or one n-directional transformation (or as a special case for two artifacts: bi-directional transformation). The problem is with unidirectional transformations that we have to develop two transformations. Sometimes we can generate the reverse direction from the original one, but it will often not produce the desired result. In contrast n-directional transformations are more flexible for problems like synchronization. The basic idea of n-directional transformations is to define mappings between the elements of the target and the source metamodel(s), and based on this mapping the transformation of reverse direction can be executed in theory. The idea of defining mappings between the elements comes from the theory of triple graph...
A correspondence graph is defined between the target and the source artifacts that defines the relationship (the mappings) between the two sides. The main problem with n-directional transformations is that the reverse direction cannot be uniquely determined in significant number of cases: too many possibilities may exist during a given transformation.

In this paper a method is presented that helps us realize the incremental model synchronization in our modeling and transformation framework. The rest of the paper is organized as follows: Section II briefly describes two standards that can execute n-directional transformations in theory. We introduce our modeling and model transformation tool, Visual Modeling and Transformation System (VMTS) [2]. The steps of our research activity in model synchronization are also described, our earlier experiments are presented. Section III describes how a newly created element can be traced back to the source artifact, without having to rebuild the whole model. Finally, Section IV summarizes the results.

II. BACKGROUND AND RELATED WORK

Graph transformation systems make use of graph rewriting techniques to manipulate graphs. A graph transformation is defined in terms of a set of production rules. A production rule consists of a left-hand side (LHS) graph and a right-hand side (RHS) graph.

Triple Graph Grammars (TGGs) were introduced in 1994 [3]. Triple graph grammar rules model the transformations of three separate graphs: source, target and correspondence. The key idea of triple graph grammars in synchronization is that during the transformation the correspondence graph stores additional information about the transformation process itself. This information is needed to propagate incremental updates of one data structure as incremental updates into its related data structures [4]. The source and the target models often do not develop simultaneously: i.e. one graph evolves and the other one has to be updated accordingly.

QVT (Query/Views/Transformations) [5] is the upcoming OMG (Object Management Group) standard for the transformation of MOF (Meta-Object Facility) [6] models. A closer review of QVT shows that parts of specification are structurally quite similar to triple graph grammars. QVT defines a standard way to transform source models into target models. Both QVT and TGGs declaratively define the relation between two models. With this definition of relation, a transformation engine can execute a transformation in both directions and based on the same definition, can also propagate changes from one model to the other.

Both of QVT and TGG are lacking implementations. The QVT is too complicated, there is currently no transformation language implementation available being 100% QVT compatible.

Visual Modeling and Transformation System (VMTS) [2] is a model-based, n-level metamodeling and transformation framework. It allows defining different metamodels, models that are instances of the metamodels and model transformations. In VMTS, model transformations are based on the double-pushout (DPO) approach [7] and comprise of rewriting rules. The execution order of the rules is defined by a control flow model.

In VMTS, we can define a rewriting rule by their left-hand side (LHS) graph and right-hand side (RHS) graph. The LHS and RHS graphs are composed of metamodel elements. Applying a rewriting rule means finding a match of LHS in the graph to which the rule is applied, and replacing the matched subgraph with RHS.

VMTS allows use of transformations when the input and the output models are the same (in-place transformations) and when they are different. First, we used in-place transformations to analyze a solution based on the undo stack concept. Our first goal was to restore the original source model from the produced artifact. An undo stack was built for the transformation procedure, and the executed operations were saved during the transformation (such as create node, remove node, edit attribute). In the reverse direction the inverse operations of the undo stack entries were executed. This type of solution could be used when the input and the output model were the same. Figure 2 shows an in-place transformation, where the executed operations are saved into an undo stack. R1, R2, R3 are the rewriting rules of the transformation, their results are respectively depicted below. The original model is drawn with a solid line and the changes are indicated with a dashed line.

To restore the original model, the undo stack can be used without any control flow. But the modifications of the target model cannot be traced back in this case. Figure 3 depicts how the original model was produced by inverting the previously executed operations.

In most of our case studies the input and output models are different, and the models coexist. Furthermore, the metamodels of the input and the output are different. The operation of the transformation can be reverted by the undo stack, but any information about the connection between the input and the output models are lost. This means that if we modify the target, the modifications cannot be recognized, and traced back to the source. Hence we do not know how the modifications should
trace back (however in this approach if we created a new node in the target model it does not trigger changes in the source model as we would expect). The undo stack is traced only for the operations of the transformation. Our next aim was to trace back a modification from the target model to the source model. We chose to trace back to the newly created nodes to the source model. We created a reverse transformation in VMTS by hand, and during its execution we used the saved trace information from the undo stack. Another problem was that the list of the saved operations are not sufficient. The information of which nodes are created from which is lost. We added some trace information to the undo stack, we saved the matched nodes and the created nodes of the transformation process (like in TGG). With this additional trace information we can track the newly created nodes from the target model to the source model. Next section presents how the newly created nodes can be traced back to the source model incrementally by model transformations.

III. USING TRACE INFORMATION IN REVERSE TRANSFORMATIONS

The model synchronization between two artifacts can be realized by bi-directional model transformations. In our approach the bi-directional model transformation is defined by two unidirectional transformations. This approach has been motivated by the following reason: processing the target artifact may require different processing steps. In many cases the sequence of rules - the structure of the control flow - may be different in the forward and the reverse direction, thus one bi-directional transformation can be very complex. (Let’s suppose the source model is processed with a depth-first-search algorithm, while it would be more practical to process the reverse direction width a breadth-first search algorithm).

Executing the model transformation raises some questions: which extra information is needed for the reverse direction to restore the original model, and how can we store the extra information? The following sections presents an algorithm that helps us realize incremental model synchronization in VMTS.

A. Storing Trace Information

We have two ways to store the extra information: locally or globally. Storing information locally means that every rewriting rule has its own storage where the extra information can be persisted (different rules save the information to different storages). This means that the rules of the reverse direction must know which storage entry contains the necessary information. The problem with this is that the control flow and the rules of the reverse direction can be totally different from the original. One rule may use information from numerous storage entries, or one storage entry can be used by many rules in the reverse direction. The transformation engine knows what sort of trace information is needed in the reverse direction, but does not know which is appropriate storage. Figure 4 depicts a control flow, and the corresponding reverse direction (the reverse transformation consists of more rules, and the different rules use trace information from the same storage).

The global storage is more useful in this case, every rule saves the extra information in a single common storage, and the rules of the reverse direction can use this storage too. Figure 5 shows the use of the common storage for each rule.

If one common storage is used for both direction, no mappings are needed between the rules of the reverse direction and the storages of the original direction. With local storages we would create mappings between them.

B. The Saved Information

The tracking information is collected during the transformation procedure, after each rule execution. The goal of saving information is to track the transformation procedure, and use it during the reverse direction process.

The tracking information consists of pairs of matched node sets - created node sets in the applied rule. The set of matched nodes and the set of created nodes are saved as a pair of sets. This means that the set of matched nodes of the applied rule and the set of created nodes of it are saved as a pair of sets. Figure 6 depicts the used data structure.

This is very similar to the TGG approach. The difference between the TGG and our approach is that in TGG the nodes
of the LHS are mapped to the nodes of the RHS, but in our case the whole match is always mapped to the newly created nodes. In many cases the generated elements depend on the whole match, and the mappings between the elements of the target and the source are unnecessary.

In the reverse direction the set of created nodes - set of matched nodes pairs facilitate identification of nodes that have been created by the transformation. We store the matched nodes and the created nodes, but in the algorithm we use only the created nodes set. The matched nodes are reserved for future development. This can facilitate to determine the attribute editions, and node deletions. Algorithm 1 describes how the trace information is saved.

**Algorithm 1: SaveTraceInfo**

1: \texttt{SaveTraceInfo}
2: \hspace{1em} \texttt{nodePairs} = new empty list of nodePairs
3: \hspace{1em} \texttt{createdNodes} = \texttt{GetCreatedNodes} \texttt{(currentRule)}
4: \hspace{1em} \texttt{matchedNodes} = \texttt{GetMatchedNodes} \texttt{(currentRule)}
5: \hspace{1em} \texttt{nodePairs}.\texttt{AddPair} \texttt{(matchedNodes, createdNodes)}

**Algorithm 2: ApplyRuleWithTrace**

1: \texttt{ApplyRuleWithTrace}
2: \hspace{1em} if \texttt{currentRule}.\texttt{IsMatch} then
3: \hspace{2em} \texttt{matchedNodes} = \texttt{GetMatchedNodes} \texttt{(currentRule)}
4: \hspace{2em} \texttt{isContains} = \texttt{storage}.\texttt{FindInCreated} \texttt{(matchedNodes)}
5: \hspace{2em} if \texttt{isContains} then
6: \hspace{3em} \texttt{currentRule}.\texttt{Apply}()
7: \hspace{1em} end if
8: \hspace{1em} end if

Trace information is collected during the forward transformation process. The reverse direction can use this information to recognize which elements are created by the transformation. These elements are tracked, and when a new element is added to the model, the reverse direction can trace it back to the original model. The original model will not be rebuilt, because the transformation engine knows the old elements from the saved trace information, and can add new elements, edges to the original artifact. The presented methods cannot be used generally. These can be used when:

- The target and source artifacts are not modified simultaneously.
- Elements are not deleted from the artifacts.
- Attributes are not modified.

Our future research moves towards three important fields: (i) Improving our methods to extend the usability. (ii) Generate the reverse transformation automatically. (iii) Reduce number of irreversible transformations.

**Acknowledgement**

The fund of ”Mobile Innovation Centre” has partly supported the activities described in this paper. This paper was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

**References**