Traceability-based Incremental Model Synchronization

ISTVÁN MADARI, LÁSZLÓ ANGYAL and LÁSZLÓ LENGYEL
Budapest University of Technology and Economics
Department of Automation and Applied Informatics
1111 Budapest, Goldmann György tér 3., HUNGARY
{istvan.madari, angyal, lengyel}@aut.bme.hu

Abstract: Model transformation is a crucial aspect of Model-Driven Software Development. With the help of model transformation, we can generate source code or other artifacts from software models. However, a recurring problem in software development is the fact that source and target models coexist and they evolve independently. In general, a modeled system is composed of several models that are often related to one another. Consequently, the related models will not be consistent anymore if one of them is altered in the development process. For that reason, a model synchronization method is necessiated to resolve inconsistency between the modified models. Performing synchronization manually can be an error prone task due to the number and complexity of model elements. In model-driven technologies, where processing is carried out as a series of model transformations, applying model transformations can also be a reasonable option for the reconciliation. This paper presents an approach that uses trace models and model transformations to facilitate incremental model synchronization.

Key–Words: Model transformation; Model synchronization; Traceability; Trace model

1 Introduction

In model-driven software development the synchronization of the affected models is a central issue, nonetheless, to the best of our knowledge, perfect model synchronization is yet an unresolved problem. Often the relationships between the different model elements are very complex, due to the fact that models to be synchronized belong to different domains. Besides, the diversities of these domains can be very serious: they can implement different abstraction levels and may conforms to very dissimilar visual language.

In the model-related technologies we often use model transformations as a generator function in order to process the implemented models. It means that the final products (source code, test cases, part of the documentation) of the model-based development can be generated by model transformations. Thus implementing the synchronization process as model transformations is an obvious demand, if the generator process implemented in the same way.

Diversities of the domains follow that, the model transformation leads to information loss, since a model can hardly ever express all information from another domain. Generally, a synchronization process finds relationships between models, checks consistency, propagates changes and resolves conflicts if necessary. However, to discover the relationships automatically between model elements of different domains is unimaginable, if the metamodels are not similar. Moreover, the consistency checking and change propagation are very difficult steps, due to the fact that there are may complex relationships between model elements of different domains.

Consequently, preserving information and taking relationships into account can facilitate successful model synchronization.

Incremental model transformation is a very effective way of model synchronization [1]. The basic idea behind incrementality is that the synchronization process does not generate the whole artifacts from the scratch, but the transformation only updates the models. The possibilities for incremental transformation are discussed in [2]. With incremental model synchronization the information can be preserved during the transformation process. Thus, if the transformation is applied between different metamodels, the original models keep the information that cannot be transformed.

The relationships between different model elements play a central role in the synchronization. In practice, the relationship between different model elements means that mappings are defined between the target and the source model elements. Hence, we will know exactly the details: which element caused the changes in the target model during the transforma-

tion. The idea of defining mappings between the elements comes from the theory of Triple Graph Grammars (TGG) [3], where a correspondence graph is defined between the source and the target artifacts that describes the relationship (the mappings) between the two sides.

With relationship information, the transformation rules can recognize the elements that should be newly created, deleted or left untouched when the transformation is executed in the synchronization process. This extra information can be automatically derived if the model transformation preserves trace information during its execution.

In our previous work [4, 5], we have defined a data structure for tracing and developed an algorithm that can be used to implement an incremental synchronization method with unidirectional model transformations. In this paper we introduce our improved approach, where trace models are used instead of trace data structures to facilitate model synchronization.

The feasibility of our approach is also illustrated via a case study: synchronizing C# syntax tree model and .NET Compact Framework user interface model.

2 Related Work and Background

Triple Graph Grammars (TGGs) were introduced in 1994 [3]. Triple graph grammar rules model the transformations of three separate graphs: source, target and correspondence graphs. 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.

QVT (Query/Views/Transformations) is the OMG (Object Management Group) standard for the transformation of MOF (Meta-Object Facility) models. A closer review of QVT shows that parts of the 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 QVT and TGG are lacking implementations. QVT is too complicated, there is currently no transformation language implementation available being fully QVT-compatible.

Visual Modeling and Transformation System (VMTS) [6, 7] is a graph-based metamodeling system. Metamodeling means that we can create models not only for predefined modeling languages, but also we can create new modeling languages as well. New languages are defined by creating models of models, called metamodels. VMTS uses n-layer, layer transparent metamodeling. This means that we can also create the metamodels of metamodels, since the base functions are the same for all modeling layers. Models and transformation rules are formalized as directed, labeled graphs, which consist of individual attributed nodes and edges. Also, VMTS is a model transformation system, which transforms models by executing graph rewriting rules [8, 9], that follow the philosophy of the double-pushout (DPO) [10, 11] approach, in an order defined by a transformation control flow model.

Source Code Modeling

In [12] we introduce a technique developed for VMTS to facilitate the production of textual content from visual models. According to that technique we have created a metamodel for the target programming language in VMTS, which conforms to the most of the grammars of these languages (referred to as *CodeDOMLight*). The elements of the metamodel also contain their specific textual representation to enable the production of source code files from this kind of instance models. The source code of the text generator that conforms to the textual concrete syntax is automatically derived from the metamodel. The code generator is a metamodel-specific text printer that emits text fragments while traversing the elements of the instance model of *CodeDOMLight*. A part of the metamodel (*CodeDOMLight*) is depicted in Figure 1. The attributes of the nodes, the statement and expression definitions are omitted to save some space.

Consequently, the source code generation in VMTS means just a syntax tree composition using model transformations. The advantage of modeling the source code files to be generated is that further model processing tasks (e.g. refactoring, model-code synchronization) can be performed.

There is a well-known file modeling and manipulation approach in the practice: the Document Object

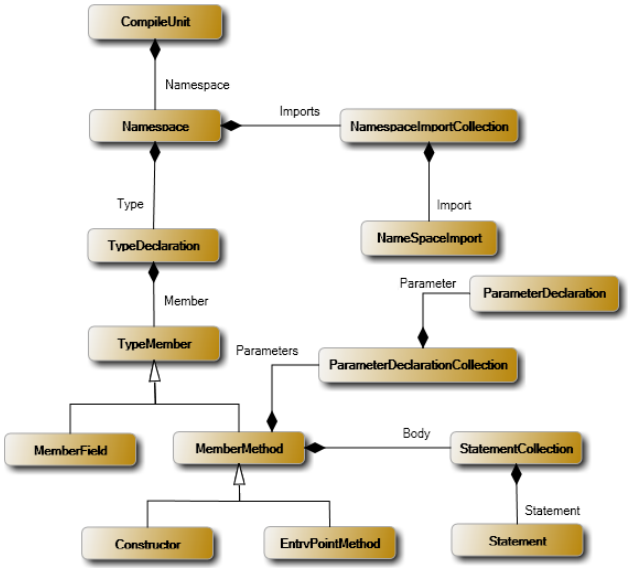

Figure 1: Structural Elements of the *CodeDOMLight* Metamodel

Model (DOM) [13] defines a standard way for accessing and manipulating XML documents. The DOM represents an XML file as a tree-structure. This approach can be abstracted and generalized in order to obtain a way for accessing and dynamic manipulating hierarchical files.

Source code files are also considered as hierarchical structures when they are processed. An abstract syntax tree (AST) is a tree data structure built during the parsing of a textual content conforming to a grammar. The parser reads the input character stream or the stream of tokens and emits a structure accordingly to the grammar. The tree is composed by nodes defined by the building blocks of the language in a hierarchical order they have matched.

There are existing commercial technologies to generate code through tree synthesis. CodeDOM is an object technology provided by Microsoft for runtime code generation in multiple programming languages. The CodeDOM library consists of a set of classes that are sufficient to describe the tree structure of an object oriented source code independently from the syntax of a specific programming language. To represent source code, from CodeDOM elements an object structure can be composed, which models the structure of the code. There are numerous built-in and third-party generators available for CodeDOM to render a synthesized object tree in the textual syntax of a supported language. The code providers of CodeDOM are generators that pretty-print the object structure.

Pretty-printing [14] performs unparsing an (abstract syntax) tree and generates typically source code. By traversing the AST and visiting every node, tokens belonging to the nodes are printed in an adequate order. The inserted white spaces and blank lines provide that the output will be well-indented, consistent with the built in predefined layouts of the pretty-printer.

The generator for *CodeDOMLight* supports the merge of artifacts modeled using this metamodel. A merge approach can be operation-based or state-based [15]. The operation-based approach requires a tool that records the committed edit operations, while the state-based approach derives the changes by comparison after they occur. The sequence of these operations is referred to as the edit script. In our approach the change propagation is based on executing the edit scripts on other artifacts to obtain the same modified state. Since source code files can be modified outside the editor that records the edit scripts, we have chosen to support the state-based merge approach. The difference analysis and change propagation between two artifacts requires that their contents are the composition of the same kind of atomic building blocks. The content of textual and visual models can be represented by trees composed of AST nodes originated in the metamodel. Consequently, all possible operations that can be performed on the models can be described by a list (edit script) of atomic AST node operations like insert, update, move, or delete. Therefore, the AST serves as an evident ground for a fine-grained incremental update.

3 Synchronization and Tracing with Unidirectional Model Transformations

A model transformation can be unidirectional or n-directional [2, 16]. In contrast to unidirectional approaches, the n-directional transformations can be executed in multiple directions. Modeling tools often use n-directional techniques for synchronization; however n-directional approaches have several implementation difficulties. Usually the reverse direction cannot be specified in conjunction with the original transformation, so that several transformation paths may exist between the given artifacts. Moreover, in many cases no reverse direction exists. Thus, there is no clear way of defining the reverse direction.

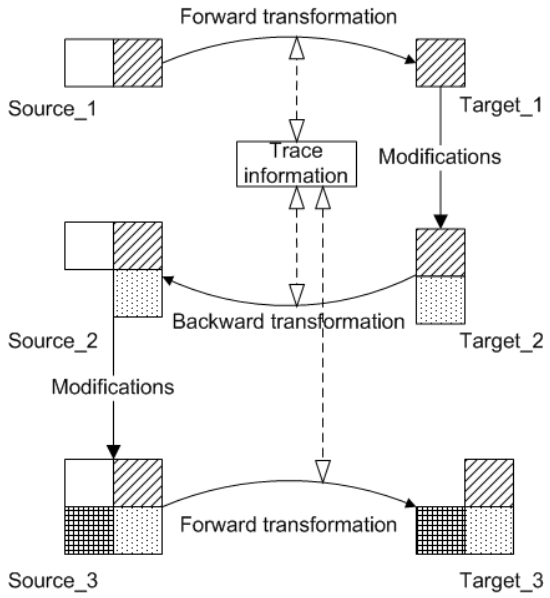

Figure 2: Model evolution during the development.

In model-driven development the outcome of a model transformation process is rarely acceptable as a perfect result, it is not the final product of a development cycle. After the transformation, modifications are required to the generated artifacts, and it follows that the source and target models need to be synchronized.

The source and target models to be synchronized are related to one another via the generation process (the first execution of the forward transformation). Trace information is saved during the transformation that contains the relations between the source and target model elements. Thus if the target models are altered, then the backward transformation process will know which elements should be updated or newly created.

As depicted in Figure 2 the forward transformation may be not suited for processing each element from the input model, because the target and the source models conform to different metamodels. The object from the source corresponding to the empty square is not present in the target model, because the forward transformation is unable to process it. Modification in the target model leads to inconsistent state between the source and the target. The backward transformation is able to update the source model with the help of the saved trace information. The backward transformation also saves trace information, thus the forward transformation can recognize which elements in the target model should be left unaltered.

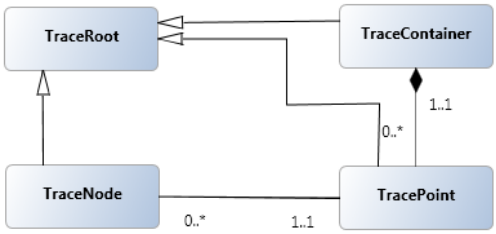

Figure 3: Trace metamodel.

As previously mentioned, during the forward transformation a loss of information has occurred. If we did not save trace information, the whole source model would be regenerated during the backward transformation, and consequently information would be irrecoverably lost.

The problem with the non-incremental update approaches is that they do not take into consideration important information from the model: the layout that is previously defined by human effort disappears. The layout in a visual model means the size of the elements and their positions, in the textual files the layout is the combination of the white spaces between the elements and the comments. The layout in both representations contains valuable extra information, which is developed into the model. Using incremental approaches the developers can work with their own models updated only in the affected parts, instead of obtaining totally new models and files after the synchronization.

4 Model-based Tracing

In [4, 5] we used a particular data structure to save auxiliary information. The rewriting rules contained additional hand-written native C# code, which persisted the given information. The saving method used file-based storage that was provided for persistency, however this approach cannot be considered as a really model-related solution.

As an improvement, we developed a metamodel, which is able to represent the auxiliary information. In Figure 3 this trace metamodel is depicted. Each model element inherits from the *TraceRoot* model element, because the inheritance helps us in the transformation development. Elements with *TraceNode* type will refer to model elements, which can be contained by other models (such as a *TraceNode* element can refer to arbitrary model element from the source or target models, irrespective of the current node type).

In details, the *TraceNode* contains a reference attribute type, which implements this relation. *TracePoint* node holds together the *TraceNodes*, and it defines a saving point that separates the trace information from each other. The *TraceContainer* element only holds together the trace elements.

The trace models can be used in addition to the source and target models, and they have to be created by the model transformation. In particular the rewriting rules have to be prepared for the additional model generation.

5 General Steps of Transformation Extension

The presented synchronization approach can be implemented as an extension of the original model transformation. The goal of the extension process is to add model synchronization capabilities to the existing model transformations, in such a way that the outputs of the original model transformations have to be the same.

In Figure 4 T and T_{sync} are model transformations; T_{sync} is derived from T by an extension process that add tracing capabilities to T; M_T, M'_T are output models while M_S, M'_S are input models; $M_T = T(M_S)$ and $M'_T = T_{sync}(M'_S)$. Consider, that the extension process does not add transformation steps to T that modifies the original output. It follows that if $M_S = M'_S$, then $M_T = M'_T$.

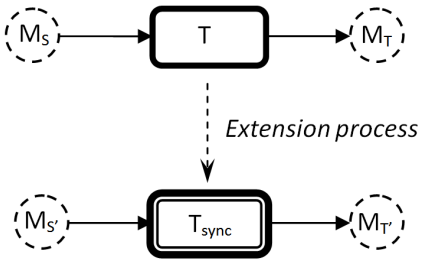

Figure 4: Extending the original model transformation

However, before the extension of the model transformation, the original control flow and rewriting rules must be analyzed to create the correct extension. In detail, the model transformation contains sequence of rewriting rules in order to (i) search (ii) create (iii) modify (iv) delete elements from the models to be processed. Each foregoing operation implemented in different way in model transformations. Accordingly, the extension process can cause very different tracing control flow. The presented synchronization algorithm does not handle node deletion and attribute modification at the moment, thus the general extension process to be presented applies to the rest of the operations.

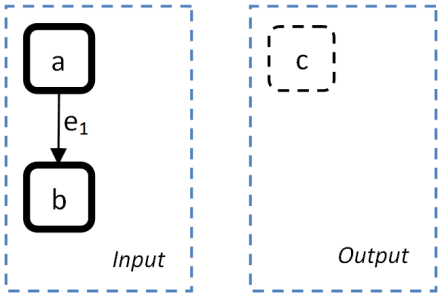

Figure 5: General *creator rule*

Extension of Creator Rules

Basically, *creator rules* search elements in the input model, and add new elements in the output model. Figure 5 depicts a general *creator rule*, that searches elements a and b in the input model with relationship e_1, and creates element c in the output model. If synchronization is needed, the creator rule must be split into 3 different rules in order to implement synchronization: (i) *Checker rule* (ii) *Tracer rule* (iii) *Synchronizer rule*. (In the following Figures, the elements with dashed border are newly created, since the solid border shows elements to be matched.)

The *checker rule* verifies the input model, and analyzes that the original *creator rule* can be applied or not. The result of a *checker rule* is a $\{true|false\}$ value that means if the rule application was succesful or not. In detail, the *checker rule* contains the original *creator rule* elements without the elements to be generated. Actually, the *checker rule* is converted from the original *creator rule* that only search elements in the input model. Figure 6 depicts the *checker rule* conversion from the original *creator rule*.

If the *checker rule* found a match in the input model, it passes the matched model elements to a *tracer rule*. The *tracer rule* checks that the founded match has a corresponding trace node in the trace model or not. If the trace model contains nodes that belong to the current match, than the founded match has already been processed, thus the transformation engine does not have to transform them again. The *trace rule* can be derived from the original *creator*

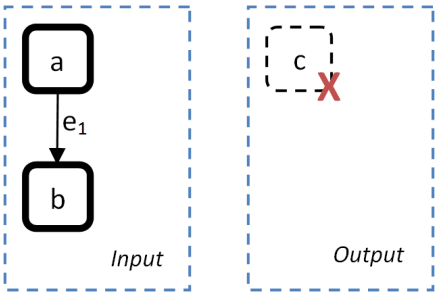

Figure 6: *Checker rule* generation

rule if we add the trace nodes to the original rewriting rule. Furthermore, the nodes to be created in the original *creator rule* have to be changed to be matched. As shown in Figure 7 the original *creator rule* is extended with *TraceContainer* (Tc), *TracePoint* (Tp), *TraceNode* (Tn) elements and *TraceLink* (Tl_x) edges. Moreover, the operation of element c is changed to *"to be matched"* instead of *"to be created"*.

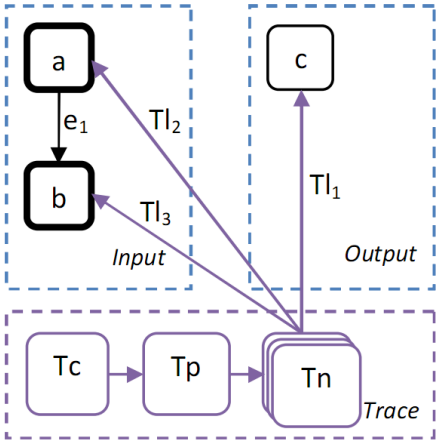

Figure 7: *Tracer rule* generation

The output of the tracer rule is also $\{true|false\}$ value that means: "the nodes to be created have already been exist in the trace model" (thus they do not have to be created again).

The aim of the *synchronizer rule* is to create nodes in the output model, and create the appropriate trace elements in the trace model. Actually, it is very similar to the *tracer rule,* although the nodes in the trace and output model are *to be created.* Figure 8 depicts the *synchronizer rule* of the original *creator rule* (in Figure 5).

The execution order of the created rules is depictied in Figure 9. In Figure 9, the original rule name is *creator rule* and three rules are generated from it. The dashed arrows mean, that if the previous rule application was not succesful the execution order will follow this edge. Since the solid arrows indicate the execution order, if the previous rule applied successfully.

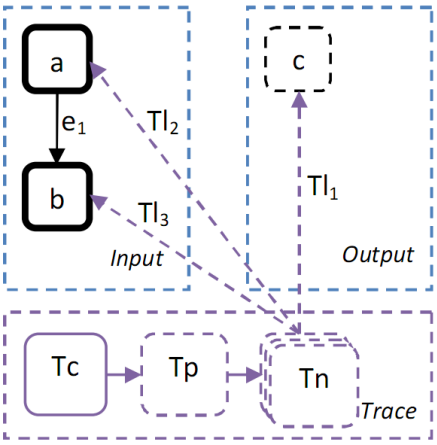

Figure 8: *Synchronizer rule* generation

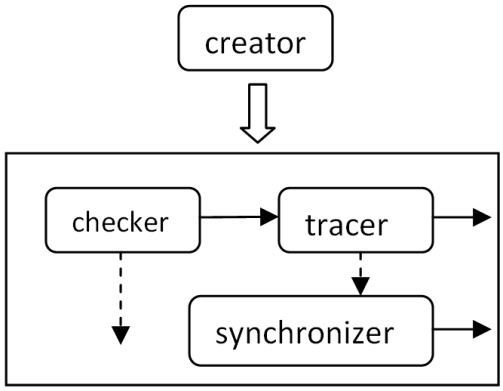

Figure 9: Execution order of *checker*, *tracer* and *synchronizer rules*

First, the *checker rule* is applied. In details, it means that the original *creator rule* can also be applied in the given input model, thus we can synchronize this transformation step. If the *checker rule* applied successfully then the execution order follows to the *tracer rule* otherwise the next rule in the control flow (through the dashed edge). *Tracer rule* checks the trace model, it searches corresponding trace nodes to the matched elements of the *checker rule*. If the *tracer rule* cannot be applied successfully, then the next rule is the *synchronizer rule*, otherwise the next rule in the control flow through the solid arrow. The *synchronizer rule* creates model elements in the input and trace models, moreover it creates traceability links between the created nodes.

6 Synchronizing Syntax Tree and User Interface Models

Currently, the development of user interfaces for three different mobile platforms is supported in VMTS [17]. To provide development environment for mobile user interfaces in VMTS, we have built metamodels for (i) the Java Micro Edition (J2ME), (ii) the .NET Compact Framework and (iii) the Symbian Series 60 software platforms. Based on these metamodels we can create user interface models in VMTS: Figure 10 depicts a .NET Compact Framework user interface model in our environment.

Figure 10: .NET Compact Framework user interface model in VMTS.

The created user interface models can be processed by model transformations as well. (In [5] we have presented model transformations that convert user interface models into different mobile platforms.)

After developing the user interface model, generating source code from the related models is an obvious demand. We have briefly described in Section 2 how the code generation can be performed in our approach from a *CodeDOMLight* model. From a user interface model, the code generation process can be completed through the following steps: (i) first, the created user interface model must be transformed into the corresponding *CodeDOMLight* model, (ii) afterwards a generated text generator (pretty-printer) will produce the source code. Figure 11 depicts an outline of the code generation and synchronization process. Actually, model transformations play a part between

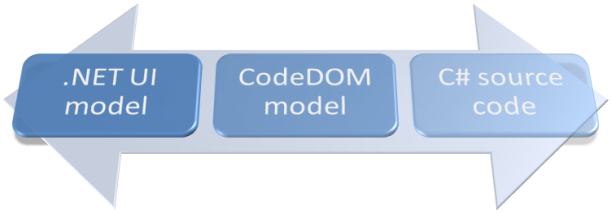

Figure 11: Code generation process.

the user interface and the *CodeDOMLight* models. The connection between the *CodeDOMLight* model and the source code is realized by a generator process that is not implemented as a model transformation.

The model transformation saves trace information, and a trace model is generated during the execution. Actually, the trace model is an additional outcome of the first execution, and it can facilitate the model synchronization. The trace information consists of pairs of matched node sets vs. created node sets in the applied rule. Let C denote the set of created nodes, and M the set of matched nodes. The set of matched nodes and the set of created nodes are saved as a pair of sets, which we denote with (C, M). In fact, it means that the transformation saves the matched nodes and created nodes for each rewriting rule into a trace model.

In the reverse direction the (C, M) pairs facilitate identification of nodes that have been created by the transformation. In practice, this means that we can create rewriting rules that check the current match against the saved nodes in the trace model. If the actual match is found in the trace model, then the current match contains nodes that were created by the forward transformation, thus the rewriting rule can indicate that the current match should be left untouched.

If the trace model does not contain elements for the current match, the transformation has to apply a rewriting rule that creates the corresponding elements and updates the original model.

Let us detail the forward transformation. The topmost element in our .NET metamodel is the *DNApplication* element. The user interface models are hierarchical models, and the transformation steps are accordant to this structure. The *DNApplication* element is processed by the *CreateForm* rewriting rule depicted in Figure 12. The *CreateForm* rule searches a *DNApplication* and a *DNForm* model elements in the input model (*Model 0* in Figure 12), then creates the appropriate *CodeDOMLight* model (*Model 1* in Figure 12).

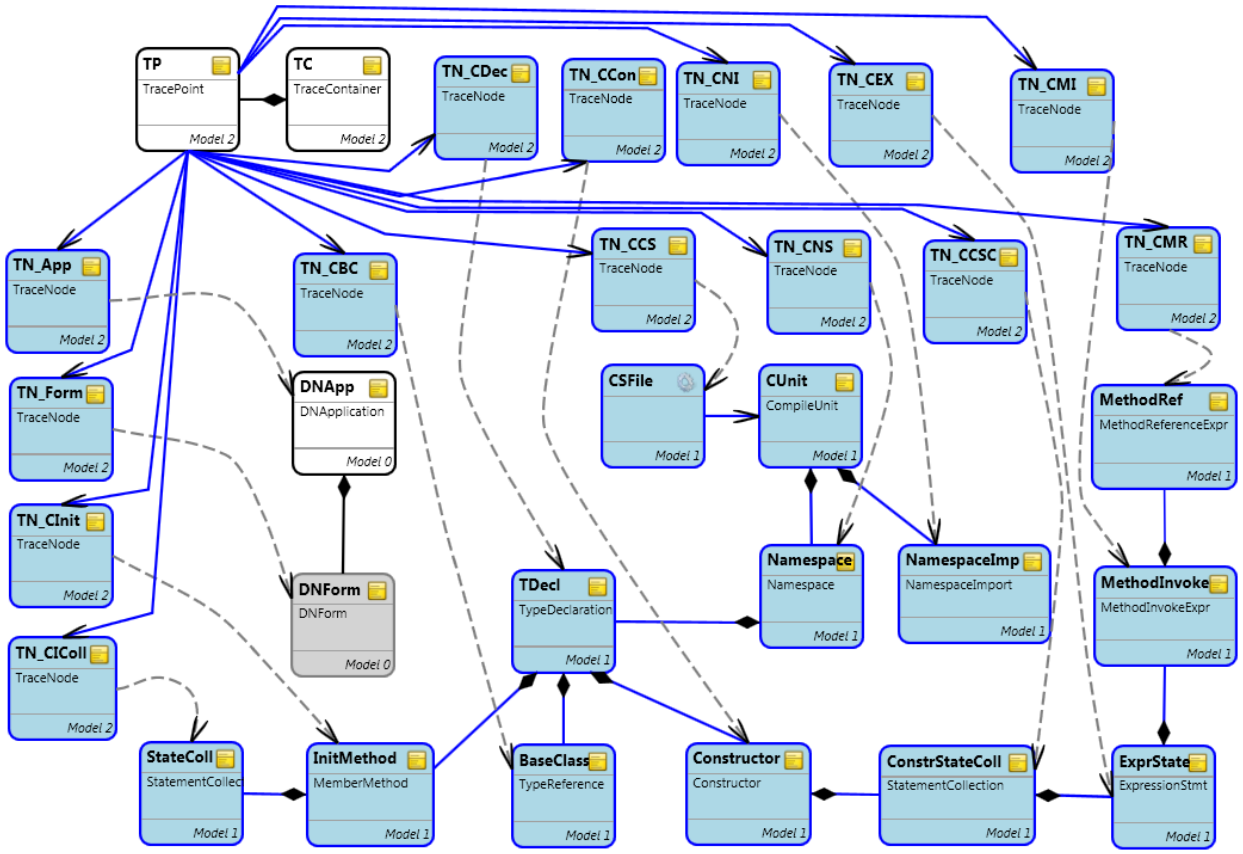

Figure 12: Create the form class, constructor, and init method with trace elements.

The created *CodeDOMLight* contains numerous elements in this rule, because C# *namespace*, *class*, *constructor* and an *initialize* method are also produced.

```
using System.Windows.Forms;
namespace VMTS.Generated {
   public class DNForm0 : Form {
        DNForm0()
        {
            InitControls();
        }
        public  void InitControls()
        {

        }
   }
}
```

Figure 13: Generated C# code.

Moreover, as illustrated in Figure 12, the trace model is created as well *(Model 2* contains the generated trace elements). Each created and matched node is related to one *TraceNode* element. In a rewriting rule the connection between the trace nodes and other model elements are represented with reference edges (dashed edges in Figure 12).

Reference edges are not real edges, the metamodel contains only a reference attribute, which behaves as an edge in the rewriting rules. Moreover, the reference edges can be also matched, created or deleted in rewriting rules. The result of the *CreateForm* rule is depicted in Figure 13.

If the source code is altered, the modifications has to be propagated back to the original user interface model. To execute the backward transformation, first of all, the *CodeDOMLight* model has to be updated. The connection between the *CodeDOMLight* and the source model is implemented as a parser application, which updates the *CodeDOMLight* model with the newly created, modified or deleted elements.

The rewriting rules of the backward transformation have to process the trace models as well. This means that the transformation has to contain rewriting rules that checks if the matched elements has a corresponding trace element or not. A backward rule depicted in Figure 14, where the rewriting rule checks whether the matched *CSFile* element is a newly created element or not.

If the rule application fails, then the matched

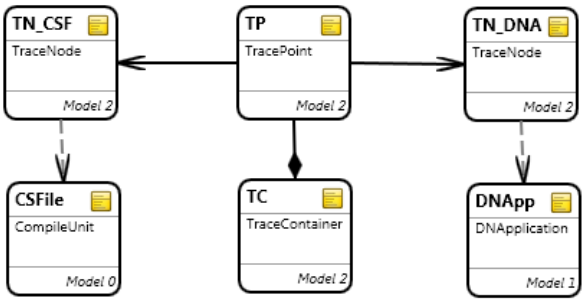

Figure 14: Rewriting rule checks the matched C# file in the trace model.

CSFile element does not have a corresponding trace node, thus it has to be processed, and a *DNApplication* element has to be created in *model 1*.

The backward transformation has to create trace elements in *Model 2*, hereby the forward transformation can also recognize which elements should be left untouched or created.

7 Conclusions

We have briefly described the feasibility of source code modeling and incremental model synchronization. This paper has introduced a model-based approach that can facilitate model transformation-based synchronization. Besides, an own source code modeling approach (*CodeDOMLight*) is presented that allows us to generate source code from a *CodeDOMLight* model, moreover it is capable of synchronizing syntax tree model with source files.

The implemented model synchronization algorithm is based on *traceability* of the model transformations. We have introduced a general tracing method (and modeling language) that can be implemented in existing model transformations. General steps to implement our tracing approach are also presented. Actually, the implementation of trace model handling leads us to a synchronizer transformation.

As a result, the user interface model is transformed into a syntax tree model and also a trace model is produced during the transformation. The generated syntax tree is processed by a code generator, which produces the final outcome.

Trace models help us discover which model elements should be left unaltered in the synchronization. Based on the trace model, both the forward and backward rules are able to propagate the changes. Although, the synchronizer model transformations contain extra rules, because the transformation has to check the related trace model as well.

However, the presented solution cannot solve yet all problems of the synchronization. Currently, the deleted nodes cannot be propagated, and the element attribute modifications cannot be tracked. We used unidirectional transformations instead of n-directional ones, thus the complete cycle needs extra work during the implementation.

Acknowledgments

The fund of ”Mobile Innovation Center” 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

[1] Holger Giese and Robert Wagner. Incremental model synchronization with triple graph grammars. In Oscar Nierstrasz, John Whittle, David Harel, and Gianna Reggio, editors, *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Genova, Italy*, volume 4199 of *Lecture Notes in Computer Science (LNCS)*, pages 543–557. Springer Verlag, 10 2006.

[2] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[3] Andy Schürr. Specification of graph translators with triple graph grammars. In *WG ’94: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163, London, UK, 1995. Springer-Verlag.

[4] István Madari, László Lengyel, and Gergely Mezei. Incremental Model Synchronization by Bi-Directional Model Transformations. In *International Conference on Computational Cybernetics*, Stara Lesná, Slovakia, November 2008.

[5] István Madari and László Lengyel. Synchronizing user interfaces of different mobile platforms. In *International IEEE conference devoted to the 150-anniversary of Alexander S. Popov*, Saint-Petersburg, Russia, May 2009.

[6] László Angyal, Márk Asztalos, László Lengyel, Tihamér Levendovszky, István Madari, Gergely Mezei, Tamás Mészáros, László Siroki, and Tamás Vajk. Towards a fast, efficient and customizable domain-specific modeling framework. In *Proceedings of the IASTED International Conference*, number 31, pages 11–16, Innsbruck, Austria, February 2009.

[7] Márk Asztalos and István Madari. Vtl: An improved model transformation language. In *Proceedings of Automation and Applied Computer Science Workshop (AACS)*, Budapest, Hungary, 2009.

[8] László Lengyel, Tihamér Levendovszky, Gergely Mezei, and Hassan Charaf. A Strict Control Flow Specification for Model Transformation. *WSEAS Transactions on Computers*, 5:390–397, February 2006.

[9] Tihamér Levendovszky, László Lengyel, and Hassan Charaf. A UML Class Diagram-Based Pattern Language for Model Transformation Systems. *WSEAS Transactions on Computers*, 4:190–195, February 2005.

[10] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, March 2006.

[11] Tihamér Levendovszky, László Lengyel, and Hassan Charaf. Extending the DPO Approach for Topological Validation of Metamodel-Level Graph Rewriting Rules. *WSEAS Transactions on Information Science and Applications*, 2:226–231, February 2005.

[12] László Angyal and László Lengyel. Synchronization of Textual and Visual Representations of Evolving Information in the Context of Model-Based Development. In *Proc. of the IEEE Eurocon 2009 Conference*, pages 438–443, St Petersburg, Russia, May 2009.

[13] Joe Marini. *Document Object Model*. McGraw-Hill, Inc., New York, NY, USA, 2002.

[14] Dereck C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(4):465–483, 1980.

[15] Tom Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[16] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA 03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[17] László Lengyel, Tihamér Levendovszky, and Hassan Charaf. Applying Multi-Paradigm Modeling to Multi-Platform Mobile Development. Technical report, October 2007.