Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam | 28

Holger Giese | Stephan Hildebrandt

Efficient Model Synchronization of Large-Scale Models

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der
Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über
http://dnb.d-nb.de/ abrufbar.

Universitätsverlag Potsdam 2009
http://info.ub.uni-potsdam.de/verlag.htm

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 4623 / Fax: 4625
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

Druck: docupoint GmbH Magdeburg

ISSN 1613-5652
ISBN 978-3-940793-84-3

Zugleich online veröffentlicht auf dem Publikationsserver der Universität Potsdam:
URL http://pub.ub.uni-potsdam.de/volltexte/2009/2928/
URN urn:nbn:de:kobv:517-opus-29281
[http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-29281]

1 Abstract

Model-driven software development requires techniques to consistently propagate modifications between different related models to realize its full potential. For large-scale models, efficiency is essential in this respect. In this paper, we present an improved model synchronization algorithm based on triple graph grammars that is highly efficient and, therefore, can also synchronize large-scale models sufficiently fast. We can show, that the overall algorithm has optimal complexity if it is dominating the rule matching and further present extensive measurements that show the efficiency of the presented model transformation and synchronization technique.

2 Introduction

For the development of a software system according to the *Model-Driven Development* (MDD) approach, different kinds of models are used, that represent different aspects of the system, which reduces the complexity of the models. But because these models are related to each other and are usually modified quite often during the development process, inconsistencies arise. Therefore, consistency between models must be also often reestablished.

Systems for model transformation and synchronization are used to automate this process. However, a simple transformation, that generates a new model from an existing one is not enough. Instead, such systems must be able to transfer modifications from a source model to a target model while avoiding to override modifications on the target model, i.e. *synchronize* the models or be incremental. An example are additional details in the target model, that cannot be reflected in the source model. Regenerating the target model would discard such details.

Furthermore, the synchronization system must often be able to transfer modifications back to the source model, i.e. the synchronization must be *bidirectional*. Here we refer to bidirectional synchronization as the possibility to transfer modifications from one model to the other model, in both directions but not at the same time. In fact, our approach supports to derive two unidirectional synchronizations from a single bidirectional transformation specification.

In case of modifications on both sides, a synchronization in both directions at the same time would be required, which also requires the resolution of conflicting modifications. As this is still an unresolved research topic [24, 19], we suggest to avoid the problem of conflicting modifications by synchronizing models *online* within interactive applications or use other means to exclude parallel modifications. An example would be the integration of two modeling tools where both tools show a model of the same system in different modeling languages. After the user modified one of the models, these changes must be quickly transferred to the other model. In an interactive application, such a process should not take longer than a second in the worst case.

The synchronisation in both directions as well as the initial model transformation must be *consistent*. Therefore, using different techniques and specifications for both directions or synchronization and transformation raises the problem to proof that the results are consistent.

Last but not least, the synchronization of models must be *efficient* to also handle large-scale models as used in practice. This is in particular true if the synchronization should happen online, where long delays are not acceptable. In our experience from the automotive domain, system models are often quite large and component-based models may contain up to 1000 components and for more detailed functional models 20,000 blocks or more are not uncommon. [8] reports a case study, where UML models from different industries with up to 36,000 elements were considered.

In this paper, we present a very efficient solution for the sketched model synchronization problem for the specific case of declarative, visual and bidirectional transformation technique of triple graph grammars (TGG) [27] that guarantees consistency as both the transformation and synchronization are derived from the same specification.

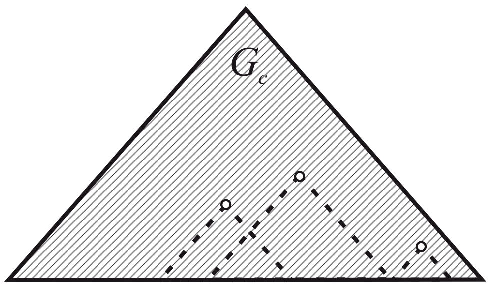


Figure 1: The batch algorithm visits the whole correspondence model.

The model transformation and model synchronization with TGGs results in a correspondence model in form of a directed acyclic graph (visualized in Figures 1 to 3 as a tree) in addition to the source and target models. This acyclic structure is employed in our new and our former solutions to guide the efficient incremental processing of changes (similar to ideas for incremental parsing as outlined in [22]). In [14, 15] we have achieved, that for most cases a single change can be processed in the average case with only logarithmical effort concerning the size of the models involved. In [12] we further improved the solution, such that even in the case of multiple changes, we can ensure only a slow increase of the efforts and that the effort always remains below or equal to the batch algorithm.

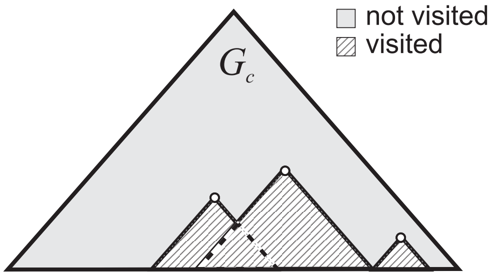


Figure 2: The incremental algorithm only visits subgraphs of the correspondence model starting at the points of modification.

However, the algorithms developed so far are still not optimal. While the batch transformation processes the whole correspondence model (cf. Figure 1) the incremental algorithms [14, 15, 12] start the synchronization at the modified elements (cf. Figure 2), thus saving effort to check those parts of the models that did not change. The improved incremental version [12] additionally sorts the modifications by their distance to the root of the model and starts the synchronization at the topmost modified element. This avoids multiple processing of elements that are affected by multiple modifications. The former algorithms always traverse the correspondence model to the leaves and synchronize structural modifications by deleting and retransforming the modified elements. Therefore, their synchronization time depends on the depth of the modification in the model, and with it, on the model's size. In the worst case, the whole model must be processed.

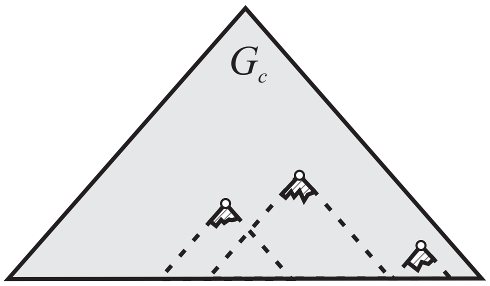


Figure 3: The new algorithm only visits those parts of the correspondence model that are directly affected by a modification.

Our analysis of the problem had revealed, however, that we can use the information available in the declarative TGG rules to also derive additional checks which can repair structural changes by adjusting links and avoiding retransformation of elements and also ensure that the effects of changes are only propagated when necessary (cf. Figure 3). Therefore, the new algorithm drastically improves our former results [14, 15, 12] and we can show, that in case the overall algorithm, and not the rule matching, dominates the complexity, it is even optimal. Furthermore, the findings are also of interest for QVT, due to the similarities between both (cf. [16]). We implemented the former as well as the new solutions for EMF models using FUJABA [29] and Eclipse.

The paper is structured as follows: We first introduce the existing model transformation and model synchronization approaches for TGGs in Section 3. Then, the potential for optimization of the former synchronization algorithms in case of changes are analyzed and the new synchronization algorithm, that excludes any unnecessary changes in the derivation, is sketched (see Section 4). In Section 5 we discuss the complexity of the problem and all presented approaches and show that the new overall algorithm has optimal complexity if it is dominating the rule matching. An evaluation of the new algorithm by comparing its performance measurements with those of former developed synchronization algorithms based on triple graph grammars as well as an ATL [21] realization of the underlying transformation problem is presented in Section 6. Finally, we discuss related work in Section 7 and provide a final conclusion and an outlook on planned future work.

3 Model Transformation & Synchronization

The triple graph grammars, we employ to perform model transformation, combine three common graph grammars, which describe how to derive a source model, a target model and a correspondence model in parallel. The correspondence model stores traceability links between corresponding source and target model elements.[1] This section outlines, how bidirectional model transformations can be derived from triple graph grammars (cf. [27]).

To illustrate the following explanations we use the example meta models in Figures 4 to 6 : Simple versions of SDL Block Diagrams (Fig. 4) are transformed into simple UML Class Diagrams (Fig. 6). Each block in the block diagram corresponds to a class with the same name in the class diagram. A special correspondence model (5) connects corresponding elements in the block and the class diagram. Note, that the block diagram is a hierarchical model, i.e. blocks are nested, while the class diagram is a flat model. All its elements are direct children of the UMLClassDiagram element.

[1]The concept of traceability links can be found in most model transformation systems, e.g. ATL [21] or QVT. It allows to easily find the target model elements that correspond to a given source model element and vice versa.

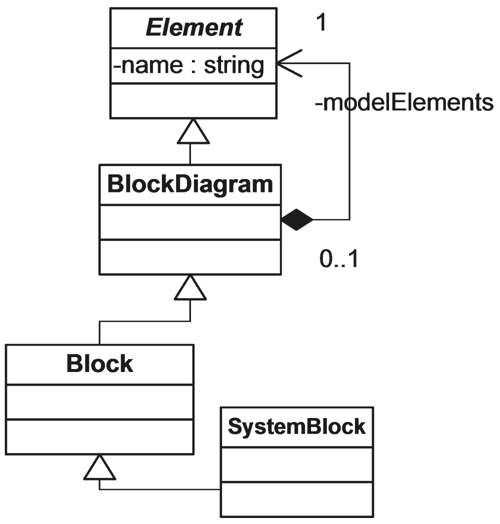


Figure 4: Block Diagram Meta Model

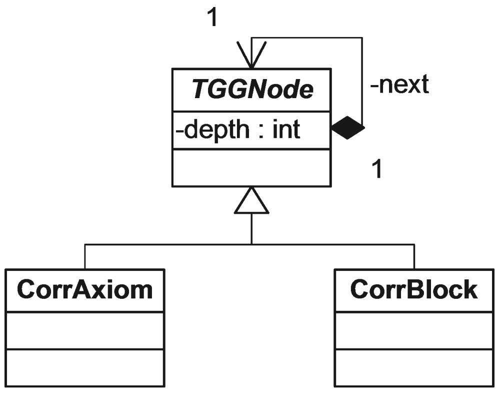


Figure 5: Correspondence Meta Model

TGGs consist of a set of rules, which create new elements depending on the application context, and an axiom, which serves as a starting point for the application of the transformation rules. Figure 7 shows an example TGG rule for the transformation of a block to a class. The rule is written in the notation used by Fujaba [29]. The left-hand-side (LHS) and right-hand-side (RHS) of a rule are combined. All elements that occur on the LHS and the RHS are colored black. These elements form the application context of the rule and are not modified during rule application. All elements that belong only to the RHS are colored green and marked with ++. These elements are created by the rule. Rules that delete elements are not used in the context of model transformation with TGGs.

A TGG rule consists of three domains. The left and right domains contain elements belonging to the source and target models respectively. The middle domain describes

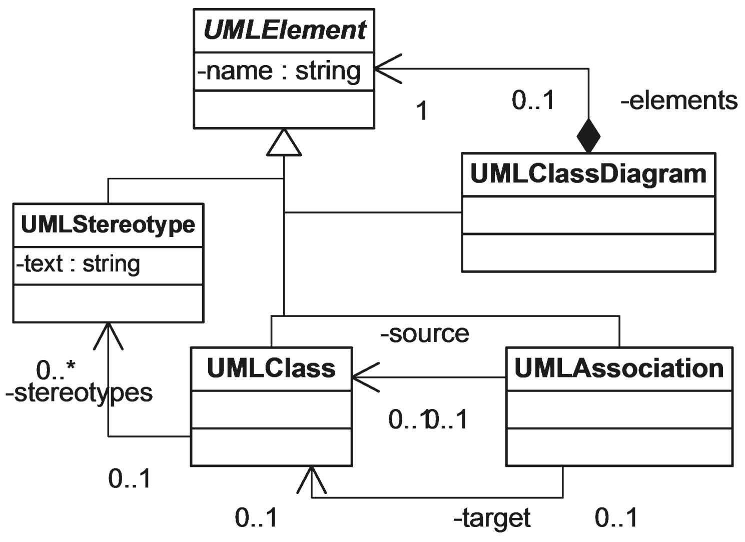


Figure 6: Class Diagram Meta Model

a transformation on the correspondence model. If all elements on the LHS of the rule can be matched to existing elements in the models, the rule can be applied and the elements of the RHS are created.

Before model transformation using TGGs can be performed, operational rules have to be derived from the declarative TGG rules. These operational rules perform the actual model transformation. Three transformation directions are possible for TGGs: Forward, backward and mapping transformations. The forward and backward transformations generate the target and correspondence models from the source model. The forward transformation defines the left domain of the TGG rules as the source model, the backward transformation uses the right domain as the source. The mapping transformation only generates the correspondence model from existing source and target models. For each of these directions another set of operational rules has to be derived. These operational rules also take care of creating the next links between correspondence nodes. These links do not appear in the TGG rules since they are always required and deriving them automatically ensures that they are consistently used and cannot be forgotten when modeling the rules.

Figure 8 shows the operational rules for the forward transformation of block diagrams to class diagrams. All elements that belong to the source domain of a rule are now also part of the application context. If all elements of the application context can be matched in the existing models, the rule can be applied, effectively transforming the source elements. One of the correspondence nodes in the application context is marked as the input node (omitted in the figure because there is only one correspondence node in the application context of any rule). This node is used as the starting point for the pattern matching process, which looks for elements in the three models that match the

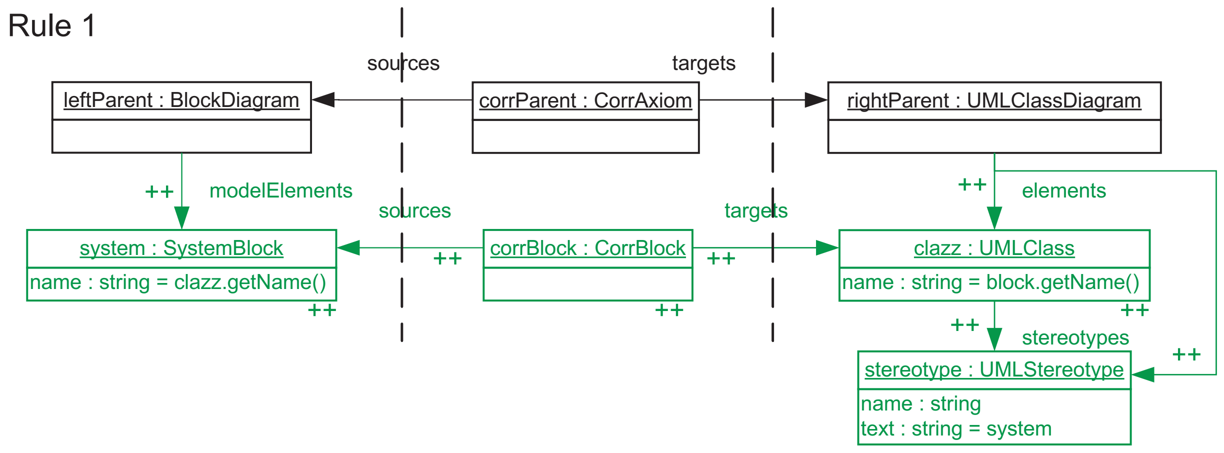


Figure 7: Example triple graph grammar

application context of the rule.

The transformation algorithm introduced in [14] consists of two parts: The operational rules, that are compiled into an Eclipse plugin, and a transformation engine, that loads the rules from the plugin and controls rule application. When performing batch transformation, the transformation engine works according to the following scheme:

```
// Batch Transformation Algorithm
Execute axiom to create root correspondence node;
Create a queue, put the root correspondence node in the
  queue;

while (queue is not empty)
{
  Remove first correspondence node N from queue;
  forall rules r that require the type of correspondence
    node N
  {
    Execute r for N in application context;
    Put direct successors of correspondence node N in
      the queue if they are not already contained;
  }
}
```

To transform a model, the axiom is executed first, which creates the root elements of the correspondence and target models. Afterwards, all rules are executed that require the type of the just created correspondence node in their application context to transform

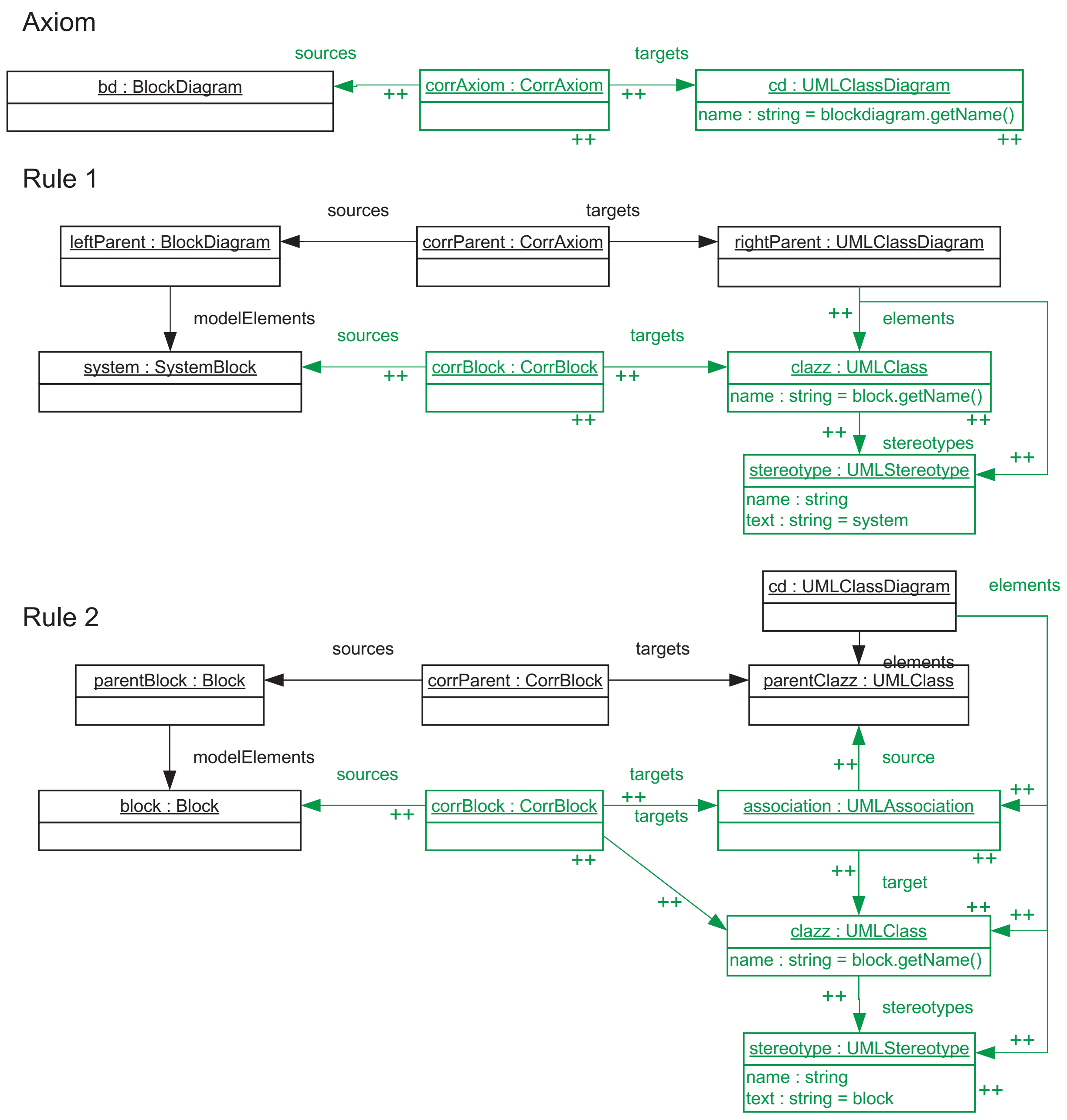


Figure 8: Operational rules derived from the triple graph grammar

additional elements. This creates new correspondence nodes. Then, all transformation rules are executed that require the type of these correspondence nodes in their application context. This process is repeated until no rule creates new correspondence nodes (cf. Fig. 9).

If the algorithm performs *model synchronization* (cf. [14]) instead of model transformation, the rules first check for modifications. In case the existing elements in the source, correspondence and target models fully comply to the rule, nothing is changed. If a modification is encountered, the rule modifies the correspondence and target models, accordingly. New source elements are simply transformed and added to the target and correspondence models. Modified attribute values are also propagated. Other structural changes, like moving an element, are synchronized by deleting and retransforming the correspondence and target elements that belong to the modified source element. Because a correspondence node is a prerequisite for its successors, deleting a correspondence node causes the deletion of all its successors and their associated target elements, as well. These nodes have to be retransformed, too. Therefore, the synchronization effort increases if a modified element has many successors. If a structural modification occurs at the top of the source model, almost the whole correspondence and target models have to be retransformed.

Instead of starting at the root correspondence node (*batch processing*), the algorithm can start the synchronization directly at the correspondence nodes belonging to the modified elements (more precisely at the parent node of these correspondence nodes), thus restricting synchronization to the affected elements and saving effort. Effectively, a batch transformation is performed on a subgraph of the model, whose root is the modified element. This *incremental* algorithm is described in [14]. To notice when a model element is modified, we use EMF's change notification mechanism. An event listener is registered at each source and target model element. If a change notification event is received, the correspondence node belonging to the modified model element and its parent node are put into the transformation engine's queue.

In case of multiple modifications, this approach can be even slower than the batch approach if multiple modifications affect the same child elements. Therefore, we improved this algorithm in [12] by sorting the queue by the depth of the contained correspondence nodes. The depth is the length of the longest path from that correspondence node to the root of the correspondence model. The *incremental algorithm with depth* starts the synchronization at the topmost correspondence node, i.e. the node with the least depth.

4 New Synchronization Algorithm

A problem, that remains unsolved, is the high worst case execution time of the synchronization algorithms (cf. Section 6). As models become larger, this time increases as well. However, interactive applications of model synchronization require quick responses to not obstruct the user too much. Potentially, even more than one synchronization steps and consistency checks might be necessary in such an interactive application in a very short time. Therefore, also the worst-case execution time of the synchronization should not be longer than about a second to guarantee such quick response times of the application.

To further decrease the synchronization time of the algorithm, all unnecessary deletion and creation of elements must be avoided. The synchronization algorithms described above always synchronize structural modifications by deleting and recreating the target model elements. In many cases, especially if elements are moved in the model, this is unnecessary. This is illustrated in the example in Figure 9. For simplicity, the UMLStereotype elements and elements links from the UMLClassDiagram to most other elements are omitted.

The block diagram on the left is initially transformed to the class diagram on the right and the correspondence model in the middle. The three models are consistent. Then the block diagram is modified. Block B2_1 is moved from its old parent B2 to the block B1. The introduced algorithms synchronize this modification by deleting the correspondence node corrB2_1, the corresponding target elements, as well as all their succeeding elements if there were any. Then these elements are retransformed. The only difference, however, are the links between the correspondence node corrB1 and corrB2_1 and the target model elements B1 and B2_1-assoc. All other elements and links in the correspondence and target model are still the same as before.

In the example, another observation can be made. The other child elements of B1 are not affected by the modification. Nevertheless, the old synchronization algorithms check them for consistency, as well as their child nodes (cf. Figure 2). If it is known, that a modification cannot affect these child elements, checking them is unnecessary. We exploit this observation and check the child elements only if a rule performed any modifications on the models that might affect these children. If a rule did not perform any modifications on a correspondence node or its associated model elements, we can be sure that its child correspondence nodes cannot be affected by that modification because the correspondence nodes form a dependency graph (cf. Figure 3). A correspondence node is always connected to all other correspondence nodes it depends on. In the example, our new algorithm starts at the node corrB2 to delete the obsolete links in the correspondence and target models. Model elements are not deleted. Then, corrB1 is the next correspondence node. Now, the links to corrB2_1 and B2_1-assoc are reestablished. This is a modification on the correspondence node corrB2_1. Therefore, it is visited subsequently. The other child node corrB1_1 cannot be affected by this modification and is not visited.

Some cases are still handled by retransforming elements. These cases are modifications that require the application of another transformation rule. Such modifications can usually not be synchronized by only adjusting some links because the element patterns that are created by different rules are usually different, too. Nevertheless, our new algorithm only deletes the correspondence node and the target model elements but not their children. Instead, the algorithm first tries to reestablish the links from the child elements to the retransformed parent elements. Only if this fails, the child elements are also deleted and retransformed. This is useful if a parent element can be transformed by several transformation rules depending on the context, but children of this element are transformed by only one rule. If the parent element must be transformed by another rule due to a modification, the child elements are not deleted but simply connected to the retransformed parent.

The correction of correspondence and target model links occurs inside the operational rules. The transformation engine of the new synchronization algorithm works as follows:

```
// New Synchronization Algorithm

// A sorted queue contains the correspondence nodes to
// be checked. The queue was filled by the notification
// event handler, that receives notifications when a
// model element is modified.

while (queue is not empty)
{
  Remove first correspondence node N from queue;

  forall rules r that require the type of
    correspondence node N
  {
    Execute r for N in application context;
    if (the rule performed any changes)
    {
      Put all direct successors in the queue that
        might be affected by those changes if they are
        not already contained;
    }
  }
}
```

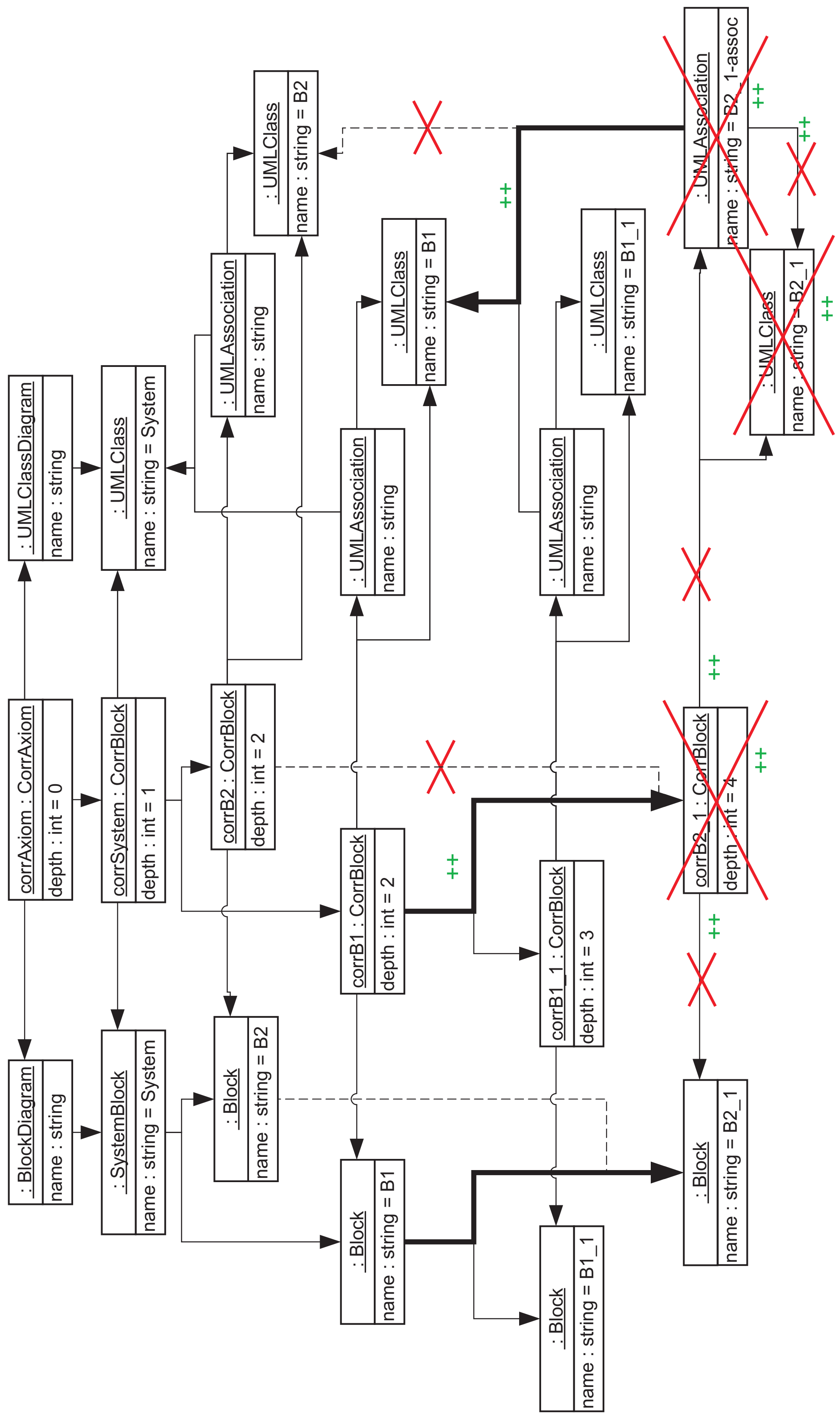


Figure 9: Example: Moving an element

5 Complexity

In order to show, that the developed overall algorithm is optimal if it is dominating the rule matching, we will discuss the complexity of the transformation and synchronization problem as well as the complexity of the presented algorithms in this section.

Rule-based algorithms for transformation and synchronization contain an overall execution scheme and the execution of the separate rules. For the rule execution holds, that the general problem behind the structural part of rule matching (for graph rewrite rules as well as other model transformation approaches such as ATL and QVT) is subgraph-isomorphism, which is known to be a NP-complete problem. Additionally specified conditions as well as updates can result in even more complex efforts (primitive recursive functions in case of OCL and Turing completeness in case of programming languages). Therefore, there can easily exist cases where a single rule, condition or update can dominate the complexity of the transformation or synchronization problem and the overall execution scheme becomes irrelevant.

However, for the overwhelming majority of transformation and synchronization specifications holds, that the conditions and side-effects over the structure and attributes are negligible, and that the efforts for the matches only grow very slowly with increasing model size.[2] Because of the manner how the correspondence nodes are related to each other in case of our approach (as well as others), not global subgraph-isomorphism but only a local search starting from the currently considered correspondence node is required to match rules. The complexity of the local search for a single match is only affected by the model size if this search has to traverse associations in the meta-model whose number is not bounded. If we have a minimum spanning tree in the graph to be matched where all edges only relate to associations which are bounded *in practice* and also the checks for the remaining edges are bounded, we can conclude that the effort for checking a match will not grow for larger models beyond the upper bound given by the combination of the bounds for the edges. In the following, we will restrict our discussion to the characterized class of transformation and synchronization problems, where we can assume that the rule matching and execution itself has constant complexity or only more or less negligible growth with respect to the model size.

When discussing the complexity of the model transformation problem, we first have to define the relevant problem characteristics. We have a source model M_1 and a target model M_2. In case of TGGs, these will finally be connected via a correspondence model M_c. Relevant characteristics for the complexity of the transformation problem related to

[2]This is true assuming also that the efforts for creating and deleting nodes and edges within the rules only growths negligibly with the overall model size. In fact, EMF does not really fulfill this requirement. However, by excluding uni-directional associations we can avoid this problem, as the later presented measurements in Section 6 will demonstrate.

the models itself are thus for $i \in \{1, 2\}$ the size of the models $n_i = |M_i|$, the size of the correspondence model $n_c = |M_c|$ or the overall size $n = n_1 + n_c + n_2$ of all three models. Note, that not only the size of source model but also the size of the resulting model is relevant here.

For the complexity of the model transformation problem holds, that we have at least complexity $O(n)$ as we have to fully traverse M_1 and have to generate M_c and M_2 thus we require at least n steps.

For the considered case, the batch algorithm for TGGs also has algorithmic complexity $O(n)$ concerning the model size as we fully traverse M_1 and generate M_c and M_2 thus we require roughly n steps (cf. Figure 1). The processing efforts for applying the rules only contributes a constant factor as the rule size, the number of rules and the effort to look for matches for a rule starting from a predecessor correspondence node can all be considered to be constant or to have constant upper bounds for the considered class of transformation and synchronization problems.

In the general case, however, the effort can be much higher. An example is the backward direction of our example where the relation between the class diagram and all classes is unbounded and grows with the model size as all classes will be connected via this association with the class diagram. However, as demonstrated in the following section, in practice, such a single association has no severe impact even for large-scale models.

To discuss the complexity of the model synchronization problem, we are interested in the changes which result for a modified source model M_1' when adjusting both the target model M_2' and correspondence model M_c' accordingly during synchronization. We thus got the size of the changes in the models $\delta_i = |\Delta(M_i', M_i)|$, the size of the changes in the correspondence model $\delta_c = |\Delta(M_c', M_c)|$ and the overall size of the change $\delta = \delta_1 + \delta_c + \delta_2$. As the synchronization result is uniquely defined, we can in fact use these characteristics to discuss the complexity of the problem as they are independent of the specific algorithm.[3]

Looking at the required effort to achieve an incremental synchronization, we can observe that we require at least $\delta_c + \delta_2$ write steps for the synchronization in order to realize the required changes to derive M_c' and M_2'. In addition, we can safely assume that we have to invest at least δ_1 steps to process the differences between M_1' and M_1, which are available as a list of modifications in the considered case. Combining these findings, we can conclude that a lower bound for the required effort for incremental model synchronization is δ and thus if we found an algorithm with computation complexity $O(\delta)$, we have found an optimal solution concerning computational complexity.

[3]Please note, that the difference between two models is measured here by the number of added or removed vertices and edges, where vertices are assumed to have an identity, while edges are only defined by the connected vertices and the edge type. Therefore, a move within a model only results in removing and adding a few edges.

The characteristics δ_1, δ_2 and δ_c are specific for the specification of the synchronization problem by a set of declarative triple graph grammar rules due to the involved correspondence model. For solutions not requiring a correspondence model or requiring only a correspondence model of smaller size M_c, the results presented in the following not necessarily apply. We can only say that $O(\delta_1 + \delta_2)$ is still a necessary lower bound. However, for all cases where δ_c is in $O(\delta_2)$ (which is only not true in rather exotic synchronization specifications) it holds $O(\delta) \approx O(\delta_1 + \delta_2)$ and thus $O(\delta)$ is still a correct lower complexity bound.

For the considered class of transformation and synchronization problems, we have shown for the incremental synchronization algorithm in [14, 15] that the effort is in $O(log(n))$ in the average case assuming a single change with a fixed number of write operations. As depicted in Figure 2 the effort mainly depends on the depth within the correspondence model.

For the version optimized for multiple changes presented in [12] follows, that the average upper complexity bound will grow from $O(log(n))$ to $O(n + \delta)$ depending on the number of changes as for multiple changes the maximal depth of all modified correspondence nodes grows and finally approaches the maximum $log(n)$.

For the complexity of the new model synchronization algorithm presented in this paper, we can make the following observations: At first the algorithm only results in write operations, which are required (where a change in the final derivation will remain) and thus the number of write operations equals $\delta_c + \delta_2$. Secondly, the number of reads is bounded by the number of initial changes in M'_1 and computed changes for M'_c and M'_2 as only changes trigger further checks. For the considered case, we can conclude that the number of checks is bounded by the number of changes. Therefore, we can conclude that also the number of read operations is in $O(\delta_1 + \delta_c + \delta_2)$. Combining our findings, we can conclude that the presented algorithm has computational complexity $O(\delta)$, and thus is in fact optimal, as this is the lower bound for the incremental synchronization problem identified in Section 5. This proves that the overall execution scheme is optimal for the considered class of transformation and synchronization problems.

If several changes in the source model result only in a bounded number of changes in the correspondence and target model, we can observe a situation as depicted in Figure 3 and thus an effort in $O(\delta_1)$ for the considered class of transformation and synchronization problems.

6 Evaluation

In this section, we present some benchmark results, that show the performance of our algorithm when transforming or synchronizing models. Our new algorithm is compared to our own old algorithms, ATL, as well as (with limitations) mediniQVT, a QVT Relational implementation [20].[4][5]

In the first benchmark, we compare the time needed to completely transform models. The benchmark tests two versions of the ATL engine[6], our old batch transformation algorithm [14] and our new algorithm. Our old incremental algorithms behave exactly like the batch algorithm if models are transformed, therefore, they are not considered here. Block diagrams with 1,000 to 5,000 blocks are generated and transformed by the algorithms (and vice versa for the reverse transformation). The transformation time is measured. Each test is repeated twenty times and the average transformation time is calculated. Note, that the meta models and transformation rules shown in Section 3 are simplified versions of those used for this benchmark. Especially the transformation rule for a block is more complex and creates seven new elements in the class diagram. Therefore, the number of elements in the class diagram is seven times as high as the number of blocks in the block diagram. The generated block diagram has the structure of a binary tree.[7] Figure 10 shows the results of this benchmark.

While the EMF ATL engine is considerably faster than the regular ATL engine, the batch and the new TGG algorithms still take less time for the transformation. As we are no ATL experts, the ATL transformation rules might be improved. Nevertheless, the benchmark shows a competitive performance of our transformation system.

MediniQVT could not be included in the full benchmarks, because it is still based on EMF 2.3 while ATL and our system already use EMF 2.4. We did some manual benchmarks by letting mediniQVT[8] transform a block diagram model with 5,000 blocks. Transforming block diagrams of that size takes about 35,000 msec on average.

[4]The transformation rules for ATL, QVT and our system, as well as the source code can be found at (provided when published).

[5]The benchmarks were run on a PC with an Intel T5500 Core2 Duo Processor with 2,5 GByte RAM and Windows XP SP3.

[6]The two versions of the ATL engine are the regular engine (package org.eclipse.m2m.atl.engine.vm) and the EMF engine (package org.eclipse.m2m.atl.engine.emfvm) from the ATL plug-in version 2.0.0.v200806101117 [1].

[7]We use this synthetic example because the two example models are typical representatives of hierarchical (elements are nested) and flat models (all elements are directly connected to the model's root) and we can use the example to check the same changes for models of different size. In addition, large-scale industry models are hard to come by in the automotive domain we mainly work in, due to non-disclosure requirements.

[8]Version 1.4.0.22283

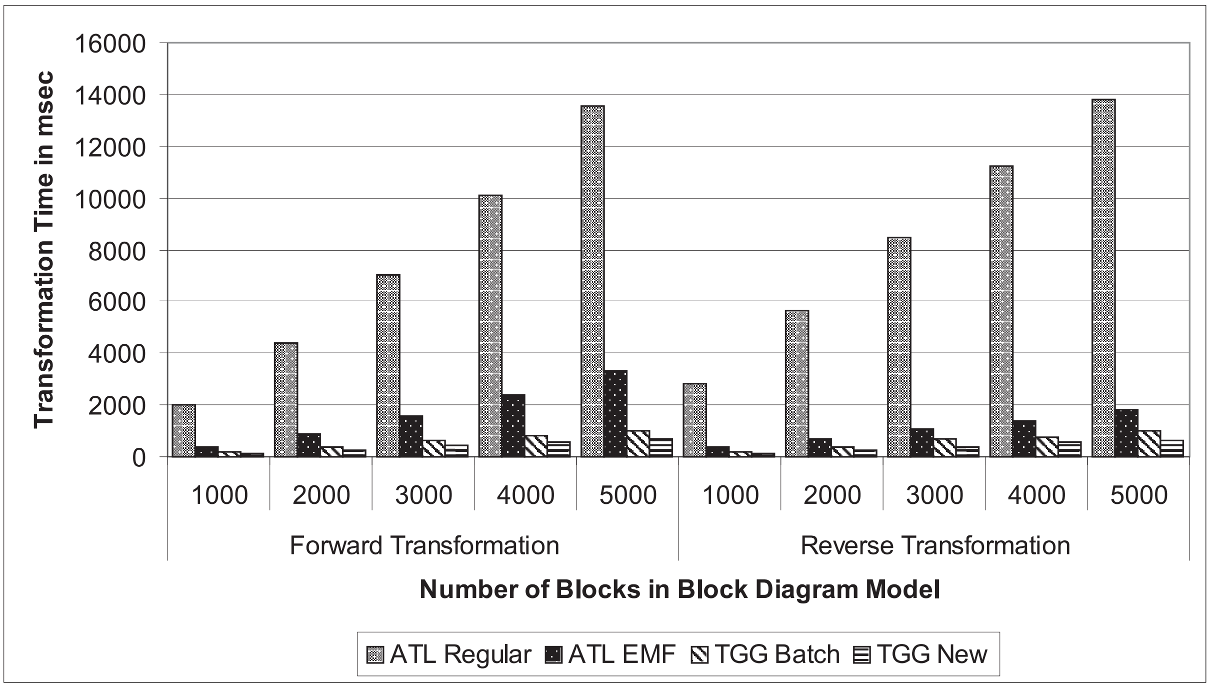
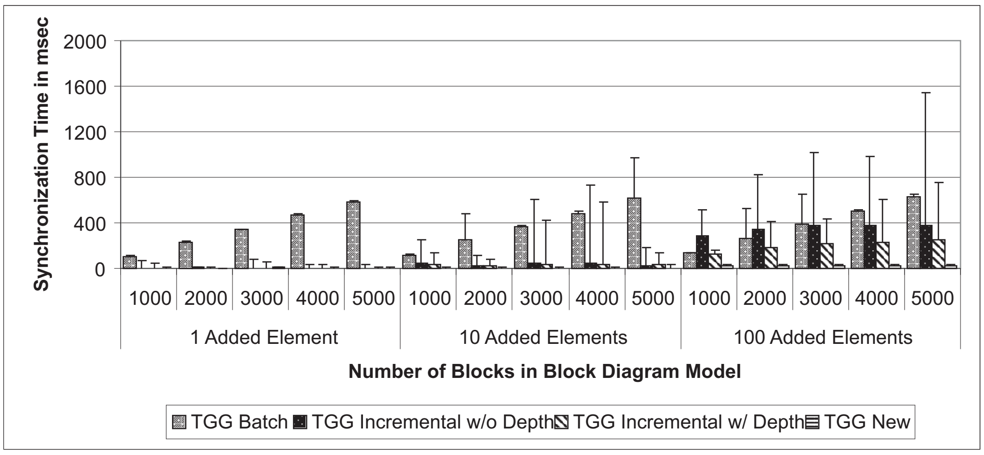


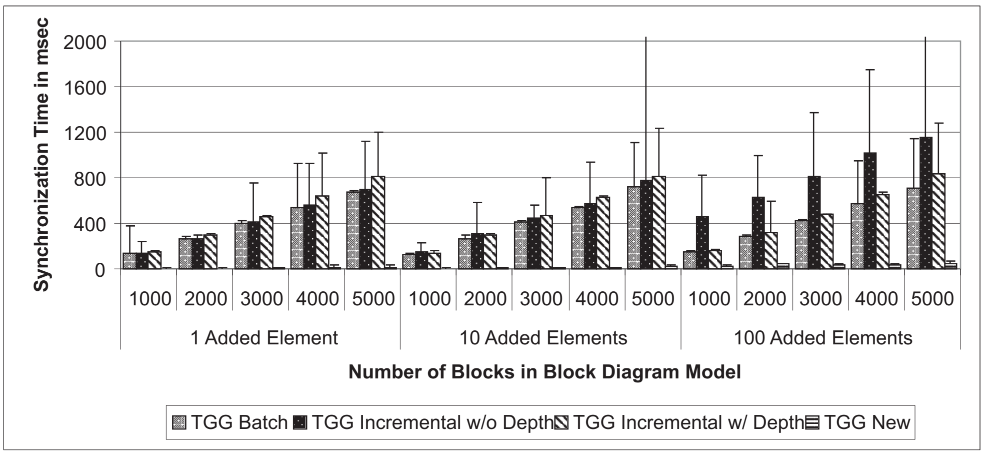
Figure 10: Average time for the transformation of models by the regular and EMF ATL engines, the batch and the new TGG algorithms

The next benchmarks test the performance when models are *synchronized*. Due to the fact, that ATL does not support model synchronization, the following benchmarks only test the batch algorithm, the incremental algorithm without [14] and with depth [12], and the new algorithm presented in this paper. The results of Figure 10 indicate, what we can expect for a complete transformation with ATL for the same problem size. An extension for ATL exists [32], that allows model synchronization. Unfortunately, we did not get it working with our example rules. However, a higher performance than for ATL alone cannot be expected, because a synchronization involves two complete model transformation and several comparison and merging steps. Model synchronization is also supported by mediniQVT, but the system always does a complete batch processing of the models. Furthermore, moved elements are synchronized by deleting and retransforming elements (like our old algorithms). Synchronizing a moved block without children in a model with 5,000 blocks takes about 125,000 msec.

For the synchronization benchmarks, block diagram models with 1,000 to 5,000 model elements are created and then completely transformed by the algorithms. Afterwards, the models are modified and the algorithms have to synchronize the models again. These modifications are adding, moving and deleting elements, and changing element attributes. These are all the elementary modifications, that can be combined to more complex modifications. The elements to be modified are chosen randomly. Every test is repeated fifty times and the time needed to synchronize the changes is measured. Note, that the system clock resolution on a Windows machine is about 15 msec. Therefore, times below that mark should be judged carefully.



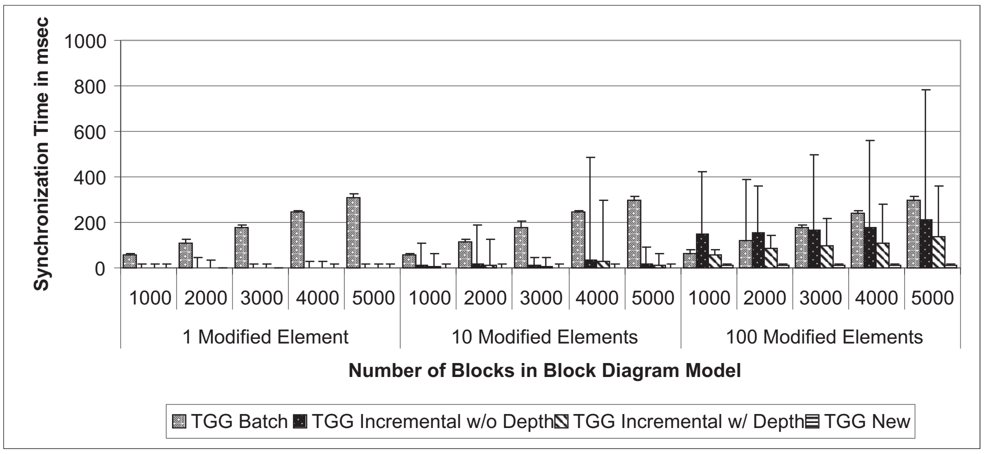
(a) Forward Synchronization



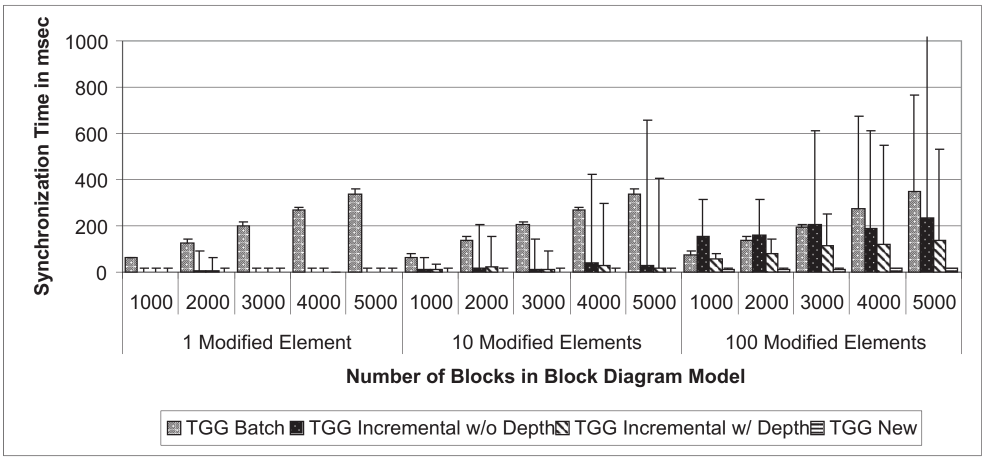
(b) Reverse Synchronization

Figure 11: Benchmark results for the synchronization of added elements.

The average and the worst measured times of the benchmarks are shown in Figures 11 to 14. The marks above the bars denote the worst measured times. In case of added blocks and changed attributes as depicted in Figure 11 and 12, the performance of the batch algorithm depends mostly on the size of the model, because the batch algorithm always traverses the whole model, regardless of the number of actual modifications. The other algorithms are influenced by the number of modified elements. The incremental algorithm without depth can get even much slower than the batch algorithm, because it synchronizes changes in the order in which they occurred. This can lead to a doubled synchronization of the same elements (see [12] for details). The incremental algorithm with depth synchronizes changes starting at the correspondence node of the topmost



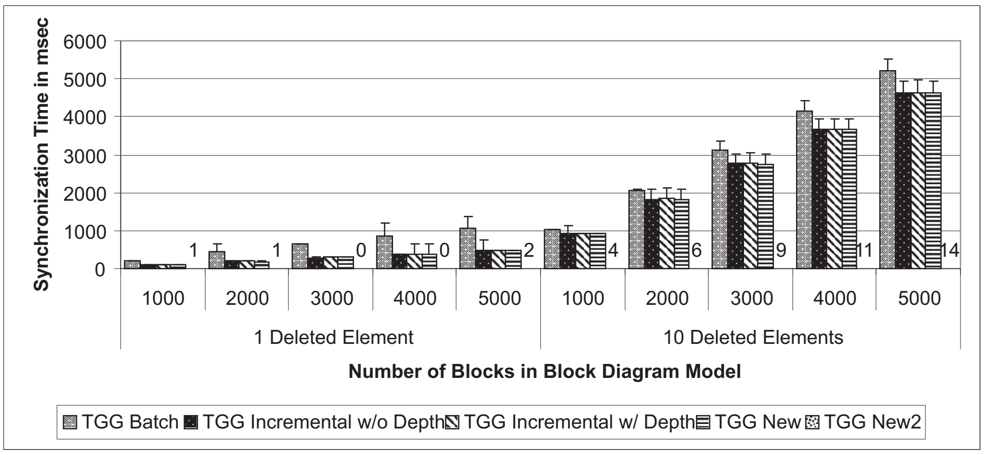
(a) Forward Synchronization



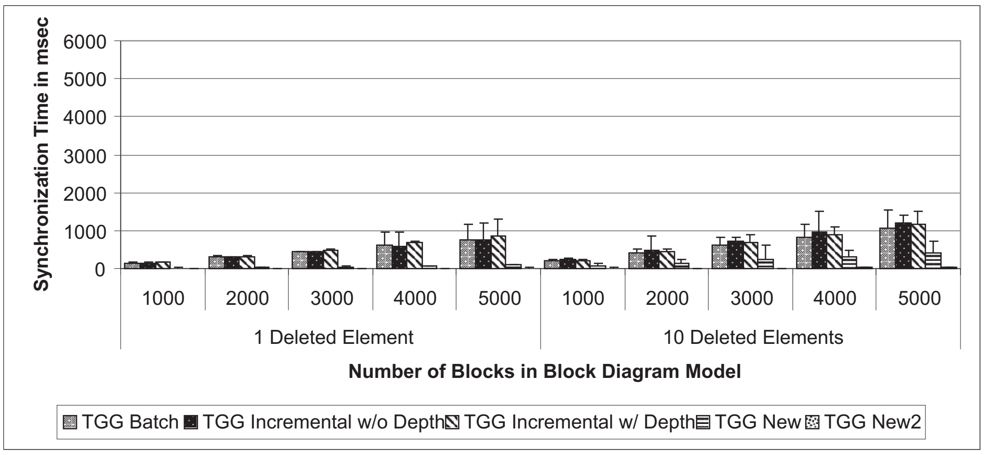
(b) Reverse Synchronization

Figure 12: Benchmark results for the synchronization of changed attributes.

modification and avoids this problem. Both incremental algorithms perform the reverse synchronization much slower. The reason is, that adding a new element two the flat class diagram model modifies the elements association of the UMLClassDiagram. This element's correspondence node is therefore put into the transformation engine's queue. This is also the root of the correspondence model, so the incremental algorithms traverse the whole model in this case. The new algorithm avoids this problem by stopping the synchronization if a modification cannot affect succeeding elements. This is also the reason for its much higher overall performance. When a new element is added to a parent element, the other children of that parent are checked by the other algorithms, as well as their children, and so on. Therefore, the synchronization effort depends on the number of successors of a modified element. In the worst case, the root element of a model is modified and the old algorithms check the whole models. The new algorithm skips these unnecessary checks, so its performance does not depend on the position of a modification in the model and its worst measured times are close to the average synchronization time.



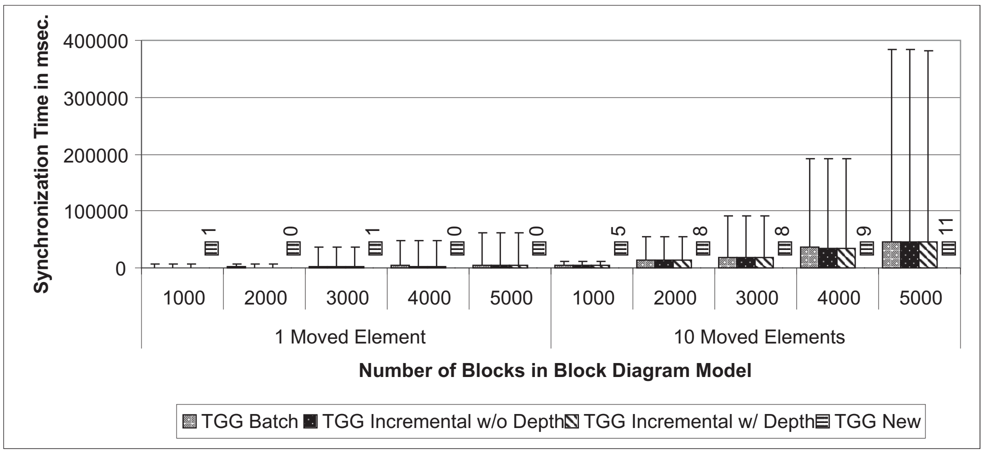
(a) Forward Synchronization



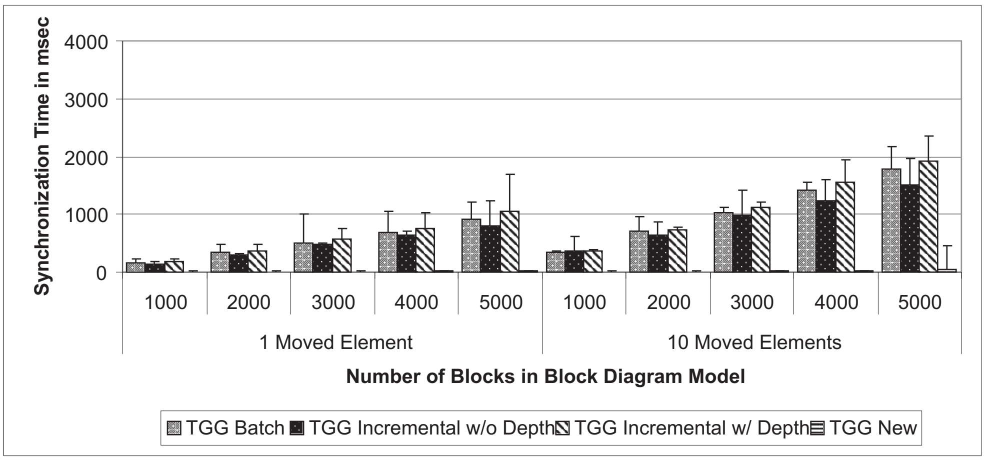
(b) Reverse Synchronization

Figure 13: Benchmark results for the synchronization of delete elements.

The deletion of elements is dominated by the execution time of the method EcoreUtil.delete() (see Figure 13), which we use to delete model elements. It ensures that an element is removed from all associations within a model, but is very time consuming. Due to the fact, that the forward synchronization has to delete more elements in the class diagram, than the reverse synchronization has to delete in the block diagram, the forward synchronization times are higher. In case of models, that contain only bidirectional associations (which applies to the example models), the deletion of all outgoing associations of an element is enough to delete it from the model and using EcoreUtil.delete() is unnecessary. This behavior is implemented in the new transformation algorithm and leads to a much higher performance when elements are deleted. For other models, the delete method has to be used, though.



(a) Forward Synchronization

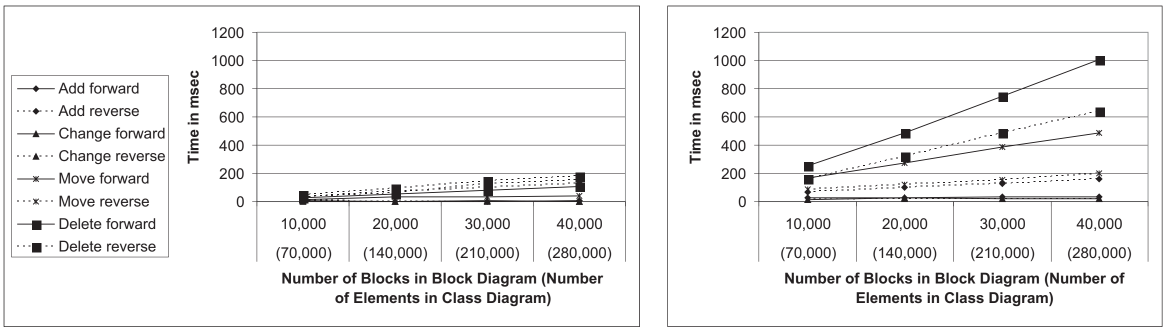


(b) Reverse Synchronization

Figure 14: Benchmark results for the synchronization of moved elements. Note the different scales.

In the benchmark, where moved elements are synchronized, (see Figure 14), the benefit of avoiding unnecessary retransformations becomes most obvious. While the old algorithms synchronize such modifications by deleting the obsolete elements (which is further slowed down by the problem described above) and retransforming the source elements, the new algorithm simply adjusts the links in the target model accordingly and avoids retransformation altogether.

Figure 15 shows the performance of the new algorithm for modifications of 10 and 100 elements and model sizes from 10,000 to 40,000 blocks. In all cases, the synchronization time remains below one second. The synchronization of changed attribute values is even independent from the model's size. The synchronization of the other types of changes is influenced by the model's size because of the unbounded elements associa-



(a) 10 modified blocks (b) 100 modified blocks

Figure 15: Average synchronization times of the new algorithm for different types of modifications.

tion in the class diagram, as described above. The deletion of elements is significantly influenced by the number of elements in the model, because the check if an element is really deleted from the model is still time consuming. The reverse synchronization of deleted and moved elements is faster, because the reverse synchronization has to handle fewer elements.

In general, the performance of the new algorithm is very good, considering that the limit of one second is only reached in case of synchronizing one hundred deleted elements in a model with 40,000 blocks (that is a total of 360,000 elements, as the class diagram contains 280,000 elements and the correspondence model 40,000 elements, because one block relates to seven elements in the class diagram and one in the correspondence model). A complete *retransformation* of a model of that size would take much longer (e.g. ATL EMF takes 134,000 msec for the forward transformation of 40,000 block.) and is therefore not practical in interactive applications. The times for mediniQVT and the ATL synchronization extension can be expected to be even higher, considering the results mentioned above.

Another effect of the proposed algorithm is, that more details in the target model are retained than before. If the target element of a moved source element contains additional details, that cannot be described in the source model, those details would be lost during synchronization with the old algorithms. The new algorithm keeps those details because the elements are not retransformed.

The presented model transformation system is already in use in some development projects with industrial partners. In the project ”Performance Assessment for Automotive Software” [18] the system is used to transform AUTOSAR [2] models to tool-specific simulation models to allow simulation of a system. In another industry project, the system is used to integrate an Eclipse-based modeling tool with another modeling tool outside of Eclipse. A special model adapter similar to that employed in [13] was developed using a COM interface to realize the communication with the external tool. Therefore, the performance constraints of the model synchronization system itself are even more demanding, in this case.

7 Related Work

An overview of model transformation and synchronization systems can be found in [6]. The paper categorizes existing approaches and explains them shortly.

As outlined in the introduction, MDD requires a bidirectional solution which preserves model contents when synchronizing as much as possible, and scales well. However, many available model transformation approaches only support classical one-way batch-oriented transformations [11]. The QVT implementations [5, 10] and the graph transformation based approaches such as VIATRA [30, 4], the GREAT model transformation system [31], AGG [9], the core PROGRES tool [26] or the core FUJABA tool [29] are only unidirectional but partly incremental. The available relational QVT implementations [20, 28] as well as BOTL [23] are bidirectional, but only support a batch-oriented processing and, thus, fail to be scalable.

Other existing TGG-based approaches also do not provide a comparable automatic and computational incremental solution (for a detailed discussion see [15]): The TGG transformation algorithm based on the PROGRES environment [3] is also incremental, but operates interactively, and is therefore inappropriate for the transformation of large models. In the incremental TGG transformation approach supported by ATOM^3 [17], updates are triggered by user actions like creating, editing or deleting elements and the specification of updates for all possible user actions is required. Thus, the consistency of the approach is difficult to guarantee and initial complete model transformations are not supported. Another TGG realization based on [29] is MOFLON [7]. It focuses on model integration and transformation for the MOF 2.0 standard [25] rather than efficient and incremental model synchronization.

Incremental model synchronization can also be seen as an inconsistency resolution problem. [8] describes an incremental solution for the related problem of inconsistency checking. The presented system allows to quickly check if a modification caused inconsistencies, and proposes solutions to the user. For a more detailed discussion of such solutions for model synchronization we refer to [15].

8 Conclusion and Future Work

In this paper, we presented our new model synchronization algorithm which dramatically improves the performance of the former introduced algorithms [14, 15, 12]. It provides the benefits of the TGG approach to be bidirectional, to guarantee consistent transformation and synchronization results and provides a much better performance, especially concerning the worst case execution time. Furthermore, the performance is only minimally dependent on the model size even if an unbounded association is present in the meta model. Our measurements demonstrate that this growth is so moderate that also large-scale models of up to 360,000 elements in our example can be handled before missing the one second worst case boundary.

In the future, we plan to further improve our results by looking into the rule execution and identified EMF bottlenecks. In addition, we want to develop a QVT compatible front-end for our approach (cf. [16]) and we want to address the synchronization of changes in both directions at the same time including techniques to cope with conflicting modifications.

References

[1] ATL Eclipse plugin project. http://www.eclipse.org/m2m/atl/.

[2] AUTOSAR Development Partnership. http://www.autosar.org.

[3] Simon Becker, Sebastian Herold, Sebastian Lohmann, and Westfechteld Bernhard. A graph-based algorithm for consistency maintenance in incremental and interactive integration tools. *Software and Systems Modeling (SoSyM)*, 6(3):287–315, September 2007.

[4] Gábor Bergmann, András Ökrös, István Ráth, Dániel Varró, and Gergely Varró. Incremental pattern matching in the viatra model transformation system. In *GRaMoT '08: Proceedings of the third international workshop on Graph and model transformations*, pages 25–32, New York, NY, USA, 2008. ACM.

[5] Borland Together Architect. http://www.borland.com/.

[6] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM System Journal*, 45(3), July 2006.

[7] Technische Universität Darmstadt. *Moflon*. http://www.moflon.org, 2007.

[8] Alexander Egyed. Fixing Inconsistencies in UML Design Models. In *Proceedings of the 29th International Conference on Software Engineering (ICSE), Minneapolis, MN, USA*, pages 292–301. IEEE Computer Society, May 2007.

[9] C. Ermel, M. Rudolf, and G. Taentzer. The AGG Approach: Language and Environment. In *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools*. World Scientific Publishing, 1999.

[10] France Telecom. *SmartQVT*. http://smartqvt.elibel.tm.fr/.

[11] Tracy Gardner, Cathrine Griffin, Jana Koehler, and Rainer Hauser. *Review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations towards final Standard*. OMG, 250 First Avenue, Needham, MA 02494, USA, 2003.

[12] Holger Giese and Stephan Hildebrandt. Incremental Model Synchronization for Multiple Updates. In *Proceedings of GraMoT'08, May 12, 2008, Leipzig, Germany*, 2008.

[13] Holger Giese, Matthias Meyer, and Robert Wagner. A Prototype for Guideline Checking and Model Transformation in Matlab/Simulink. In Holger Giese and Bernhard Westfechtel, editors, *Proc. of the 4th International Fujaba Days 2006, Bayreuth, Germany*, volume tr-ri-06-275 of *Technical Report*, pages 56–60. University of Paderborn, 2006.

[14] Holger Giese and Robert Wagner. Incremental Model Synchronization with Triple Graph Grammars. In *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Genova, Italy*, volume 4199 of *LNCS*, pages 543–557. Springer, October 2006.

[15] Holger Giese and Robert Wagner. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling*, Mar 2008.

[16] Joel Greenyer and Ekkart Kindler. Reconciling TGGs with QVT. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2007, September 30 - October 5, Nashville, USA, 2007*, volume 4735 of *LNCS*, pages 16–30. Springer, October 2007.

[17] Esther Guerra and Juan de Lara. Event-Driven Grammars: Towards the Integration of Meta-Modelling and Graph Transformation. In *International Conference on Graph Transformation (ICGT'2004)*, volume 3265 of *LNCS*, pages 54–69. Springer, 2004.

[18] Hasso Plattner Institute, Potsdam. *"Performance Assessment for Automotive Software" project website*. http://www.hpi.uni-potsdam.de/giese/projekte/permas.html?L=1.

[19] Thomas Hettel, Michael Lawley, and Kerry Raymond. Model synchronisation: Definitions for round-trip engineering. In A. Vallecillo, J. Gray, and A.Pierantonia, editors, *International Conference on Model Transformation (ICMT'08)*, volume 5063 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2008.

[20] ikv++ technologies ag. *medini QVT*. http://www.ikv.de, 2007.

[21] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. Atl: a qvt-like transformation language. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 719–720, New York, NY, USA, 2006. ACM.

[22] J.-M. Larchevêque. Optimal incremental parsing. *ACM Trans. Program. Lang. Syst.*, 17(1):1–15, 1995.

[23] Frank Marschall and Peter Braun. Model Transformations for the MDA with BOTL. In *Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications*, Technical Report TR-CTIT-03-27, Univeristy of Twente, June 2003.

[24] Tom Mens. A State-of-the-Art Survey on Software Merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, 2002.

[25] Object Management Group. *Meta Object Facility (MOF) 2.0 Core Specification*, January 2006. Document ptc/06-11-01.

[26] A. Schürr, A. J. Winter, and A. Zündorf. The PROGRES Approach: Language and Environment. In *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools*, pages 487–550. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[27] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94*, volume 903 of *LNCS*, pages 151–163, Herrsching, Germany, June 1994. Springer.

[28] Tata Consultancy Services. *ModelMorf*. http://www.tcs-trddc.com/ModelMorf/index.htm, 2007.

[29] University of Paderborn, Germany. *Fujaba Tool Suite (http://www.fujaba.de/)*.

[30] Dániel Varró, Gergely Varró, and András Pataricza. Designing the Automatic Transformation of Visual Languages. *Science of Computer Programming*, 44(2):205–227, August 2002.

[31] Attila Vizhanyo, Aditya Agrawal, and Feng Shi. Towards Generation of Efficient Transformations. In *Generative Programming and Component Engineering: Proceedings of the Third International Conference (GPCE), Vancouver, Canada*, volume 3286 of *LNCS*, pages 298–316. Springer, October 2004.

[32] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards Automatic Model Synchronization from Model Transformations. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 164–173, 2007.