Incremental Model Synchronization with Triple Graph Grammars*

Holger Giese and Robert Wagner

Software Engineering Group,
Department of Computer Science
University of Paderborn,
Warburger Str. 100,
D-33098 Paderborn, Germany
[hg|wagner]@upb.de

Abstract. The advent of model-driven software development has put model transformations into focus. In practice, model transformations are expected to be applicable in different stages of a development process and help to consistently propagate changes between the different involved models which we refer to as model synchronization. However, most approaches do not fully support the requirements for model synchronization today and focus only on classical one-way batch-oriented transformations. In this paper, we present our approach for an incremental model transformation which supports model synchronization. Our approach employs the visual, formal, and bidirectional transformation technique of triple graph grammars. Using this declarative specification formalism, we focus on the efficient execution of the transformation rules and present our approach to achieve an incremental model transformation for synchronization purposes. We present an evaluation of our approach and demonstrate that due to the speedup for the incremental processing in the average case even larger models can be tackled.

1 Introduction

Model-Driven Development (MDD) and the *Model-Driven Architecture* (MDA) [1] approach in particular have put models and model transformations into focus. The core idea is to move the development focus from programming languages code to models and to generate the implementation from these models automatically. The aim is to increase the development productivity and quality of the software system under development.

However, the modeling of large and complex software systems incorporates many informal and semi-formal notations describing the system under construction at different levels of abstraction and from different, partly overlapping view

* This work was developed in the course of the Special Research Initiative 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

points. The usage of different levels of abstraction and the separation of concerns on the one hand reduce the complexity of the overall specification, but on the other hand the increasing number of used models very often leads to a wide range of inconsistencies [2].

A possible way to face this problem is to use model transformation technology where a source model is transformed into a target model by applying a set of transformation rules. This ensures that the overlapping parts and mappings between these models are captured by the transformation itself. Unfortunately, the development of a software system is a quite iterative process with frequent modifications to the involved models. Therefore, frequent model synchronization steps are required.

Due to the size of complex models, model transformation approaches which require recomputing the transformation even though only a small fraction of the model has been modified do not scale very well. Additionally, retransforming a model each time the model evolves is not practical since refinements in more detailed target models are lost when applying a transformation from scratch. To keep the overall specification consistent after an initial model transformation, changes of one model have to be propagated in a non-destructive manner to the interrelated model by means of model synchronization. In the programming language domain, modular compilation and even incremental compilation and binding have been introduced to cope with large projects. The same problems have to be managed in the case of MDD and MDA.

We believe that sufficient tool support for incremental model synchronization by means of incremental model transformations with update propagation is a crucial prerequisite for the successful and effective application of the model-driven engineering paradigm in many cases. However, most model transformation approaches do not fully support such requirement today and focus only on classical one-way batch-oriented transformations [3].

In this paper, we present our approach for the incremental model synchronization which employs the visual, formal, and bidirectional transformation technique of triple graph grammars [4]. We will outline how this declarative specification formalism can be employed to achieve an efficient, incremental execution of the transformation rules by exploiting the known dependencies between the transformation rules.

The remainder of this paper is organized as follows. In Section 2 we first introduce a model transformation example from the area of flexible manufacturing systems which is then used to give a brief and informal introduction to the concepts of triple graph grammars. In Section 3 we explain our strategy for an efficient and incremental application of triple graph grammar rules. The evaluation results follow in Section 4. Related work and its limitations concerning the requirements for model synchronization are discussed in Section 5. The paper closes with some final conclusions and an outlook on future work in Section 6.

2 Model Transformation Approach

In this section, we first introduce a simple example which exemplifies the need for model synchronization from the area of flexible production systems. It serves then as a running example for explaining the used model transformation technique of triple graph grammars and its extensions for incremental model transformations.

2.1 The Example

In the ISILEIT project [5], we explored the possibilities of modern languages concerning their usefulness for the specification of flexible and autonomous production control systems. For the specification of the control software, we combined subsets of the Specification and Description Language (SDL) [6] and the Unified Modeling Language (UML) [7] to an executable graphical language [5]. For this purpose, a SDL block diagram is used to specify the overall static communication structure where processes and blocks are connected by channels and signal routes. This block diagram is transformed to an initial UML class diagram which can be refined and extended to an executable specification. In Fig. 1, a simple block diagram and the class diagram which results from a correct transformation are presented.

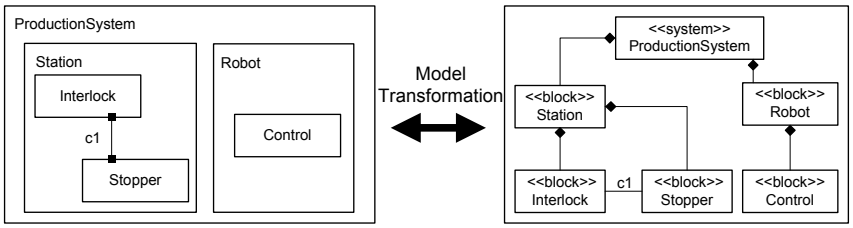

Fig. 1. Application example

Basically, systems, blocks, and processes of a block diagram are transformed to classes with corresponding stereotypes. For example, the block *Station* is represented by the class *Station* with a stereotype ≪block≫, the system *ProductionSystem* as a class with the stereotype ≪system≫. The hierarchical structure of a block diagram is expressed by composition relations between the respective classes in the class diagram. The channels and signal routes of the block diagram are mapped to associations between the derived classes. In addition, each signal received by a process in the block diagram is mapped to a method of the corresponding class in the class diagram (not shown in this example).

In order to support an iterative development process without any restrictions on the order of design steps, we allow the engineer to move freely between the block and class diagram to refine and adapt both models towards the final design. In this scenario, we have to ensure that the overlapping parts and mappings

between the interrelated models stay consistent to each other. Moreover, we do not want to override existing structures since both models can contain manual modifications and refinements which should be preserved if possible. For this model synchronization we use bidirectional and incremental model transformations based on triple graph grammars.

2.2 Triple Graph Grammars

From the previous example, it is clear that we need bidirectional model transformations. In order to support bidirectional model transformations, we use triple graph grammars. In this section, we cannot discuss triple graph grammars in full detail. Rather, we will explain the basic concepts by the help of our example and refer to [4] for a formal definition.

In order to explain the specification technique of triple graph grammars for model transformation, we have to take a closer look at the metamodels of the block and class diagram as well as at an additional correspondence metamodel needed for triple graph grammars. A metamodel defines the abstract syntax and static semantics of a modeling language. In Fig. 2, the metamodels of the block diagram, the class diagram, and the correspondence metamodel are shown.

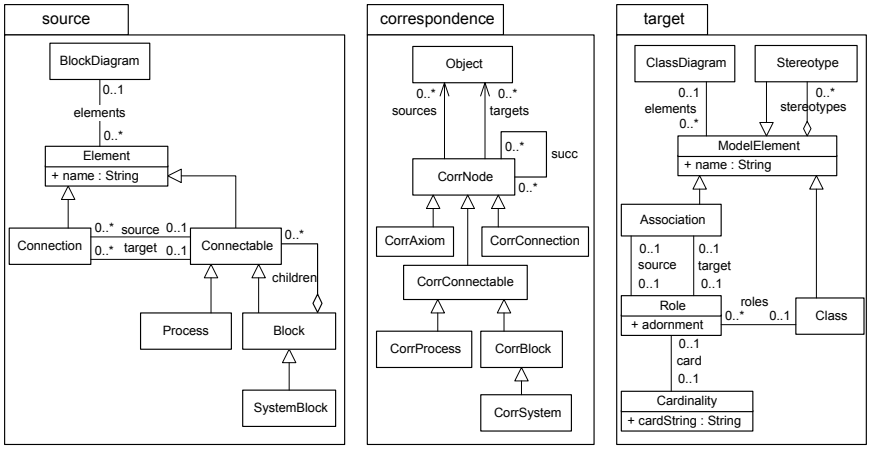

Fig. 2. Simplified metamodels of the source, correspondence, and target model

In the simplified metamodel for block diagrams, a *BlockDiagram* contains different *Elements*. An *Element* is either a *Connectable* element or a *Connection* between *Connectable* elements. A *Connectable* element is either a *Process* or a *Block*. A *Block* is a container for other *Blocks* and *Processes*. A *SystemBlock* is a special *Block* and acts as a root container for other *Blocks*. The simplified metamodel for class diagrams defines *Classes*, *Associations* and *Stereotypes*. An *Association* is connected to a *Class* by a source and target *Role* that has a *Cardinality*. A *Stereotype* can be attached to any *ModelElement* in the *ClassDiagram*.

For the specification of a triple graph grammar, we need an additional correspondence metamodel. It is shown in the middle of Fig. 2. The metamodel defines the mapping between a source and a target metamodel by the classes *CorrNode* and *Object* and its associations *sources* and *targets*. Since all classes inherit implicitly from the *Object* class (not shown here), the correspondence model stores the traceability information needed to preserve the consistency between two models. In addition, the class *CorrNode* has a self-association *succ* which connects the correspondence nodes with their successor correspondence nodes. This extra link is used by our transformation algorithm.

The two described classes and their associations are essential for our transformation algorithm. However, further correspondence nodes and refined associations can be added. In our example, we have added six additional correspondence nodes, including the correspondence node *CorrBlock* used in our example rule (cf. Figure 3). The additional correspondence classes increase the performance of our transformation algorithm since for a given correspondence node type only those rules have to be checked that have the same correspondence node type on their left-hand side.

Given this three metamodels, a triple graph grammar for our example model transformation can be specified. A triple graph grammar specification is a declarative definition of a bidirectional model transformation. In Fig. 3, a triple graph grammar rule in the FUJABA-notation is shown. Note that the vertical dashed lines are not part of the rule - they are only shown for a better understanding of the following rule description.

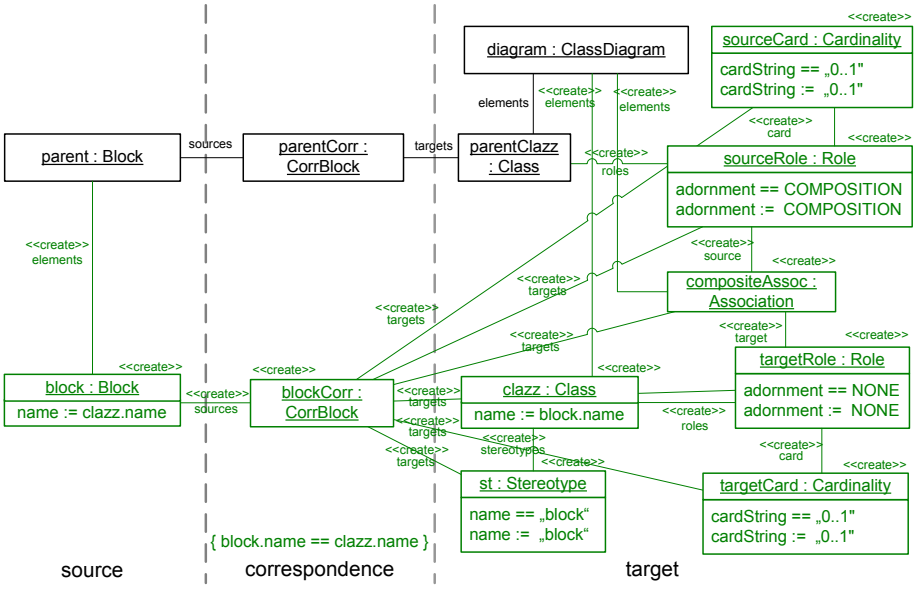

Fig. 3. A triple graph grammar rule mapping blocks to classes

The rule specifies a consistent correspondence mapping between the objects of the source and target model. In particular, the presented rule defines a mapping between a block and a corresponding class. The objects of the block diagram are drawn on the left and the objects of the class diagram are drawn on the right. They are marked with the ≪left≫ and ≪right≫ stereotypes respectively. The correspondence objects in the middle of the rule are tagged with the ≪map≫ stereotype.

The rule is separated into a triple of productions (source production, correspondence production, and target production), where each production is regarded as a context-sensitive graph grammar rule. A graph grammar rule consists of a left-hand side and a right-hand side. All objects which are not marked with the ≪create≫ stereotype belong to the left-hand side and to the right-hand side; the objects which are tagged with the ≪create≫ stereotype occur on the right-hand side only. In fact, these tags make up a production in FUJABA's graph grammar notation.

The production on the left shows the generation of a new sub block and linking it to an existing parent block. The production on the right shows the addition of a new class and stereotype and its linking to the class diagram. Moreover, to reflect the containment of the sub block, a composition association is created between the classes representing the parent block and the sub block. For this purpose, the rule contains additional objects representing the roles and cardinalities of the association. The correspondence production shows the relations between a block and a class and an additional constraint $\{block.name == clazz.name\}$ specifies that the block and the class have to be named uniquely.

Up to this point, the assignments and constraints to the object attributes have not been considered yet. Since triple graph grammars can be executed in both directions, the attribute constraints help to identify the objects to be matched, whereas the attribute assignments are applied only to created objects. However, since the computations of the attribute values may be more complicated than in our simple example, the assignments cannot be always derived from the constraints and have to be specified explicitly.

A graph grammar rule is applied by substituting the left-hand side with the right-hand side if the pattern of the left-hand side can be matched to a graph, i.e., if the left-hand side is matched all objects tagged with the ≪create≫ stereotype will be created. Hence, our example rule, in combination with additional rules covering other diagram elements, can generate a set of blocks along with the corresponding classes and associations in a class diagram. Though the transformation will not be executed this way, conceptually, we can assume that whenever a block is added to the block diagram, a corresponding class with an appropriate association will be generated in the class diagram. This way, the triple graph grammar rules define a transformation between block diagrams and class diagrams.

The correspondence production in the middle of the rule enables a clear distinction between the source and target model and holds additional traceability information. This information can be used to realize bidirectional and incre-

mental model transformations that helps to propagate changes between related models. To ensure a unique transformation we require that the set of rules is unambiguous. The complete specification of the triple graph grammar for model transformation between block and class diagrams comprises ten rules.

3 Incremental Model Transformations

While triple graph grammars are in theory a natural choice for the realization of bidirectional model transformation, in practice the required graph pattern matching is quite complex and can lead to serious performance problems if no additional information to guide the graph pattern matching is available. Since this is the most crucial part of our model transformation approach, we show in the following a practicable and efficient solution to this problem. This is achieved by an incremental transformation approach based on an analysis of dependencies between the transformation rules. However, before we attempt to overcome the limitations of the classical batch-oriented transformation approach, we identify the nature of the transformation problem in its incremental form.

3.1 Terminology

Given two sets of possible models $\mathcal{M}_1$ and $\mathcal{M}_2$, a *unidirectional model transformation* is a total function $trans : \mathcal{M}_1 \rightarrow \mathcal{M}_2$ where $trans$ can be directly computed.

Given a model $M_1 \in \mathcal{M}_1$, its transformation $M_2 = trans(M_1)$, and an arbitrary modification $mod_1 : \mathcal{M}_1 \rightarrow \mathcal{M}_1$, an *incremental model transformation* would allow to derive a modification $mod_2 : \mathcal{M}_2 \rightarrow \mathcal{M}_2$ such that it holds: $trans(mod_1(M_1)) = mod_2(M_2)$.

Assuming that the required information about the mapping between M_1 and M_2 is encoded into a mapping map_{M_1,M_2},[1] we then require that functions inc_{mod} and inc_{map} exists which can be directly computed such that

$$mod_2 = inc_{mod}(mod_1, M_1, M_2, map_{M_1,M_2}) \text{ and}$$

$$map_{M_1',M_2'} = inc_{map}(mod_1, M_1, M_2, map_{M_1,M_2}).$$

If the required effort is in $O(|\, mod_1 \,|)$ and thus proportional to the size of the modification mod_1 denoted by $|\, mod_1 \,|$ rather than the size of the model $|M_1|$, we name this a *fully incremental* solution. It is to be noted, that this optimal case that mod_2 only depends on mod_1 and not on M_1, M_2, and map_{M_1,M_2} is usually not given. Instead, at least a small fraction of M_1 and M_2 has usually to be taken into account. To make an incremental processing advantageously, the effort to determine mod_2 and compute $mod_2(M_2)$ should be much less then compute $trans(mod_1(M_1))$ in the average case. We thus call a solution *effectively*

[1] If no such mapping information is required, we can simply consider an empty map_{M_1,M_2}.

incremental if the *speedup* results in a reasonable decoupling from the model size (e.g., logarithmic effect only).

If we look at the opposite direction of the transformation, it is to be noted that the codomain $\mathcal{M}_2^* = \{M_2 | \exists M_1 \in \mathcal{M}_1 : trans(M_1) = M_2\}$ of $trans$ is not necessarily equal to $\mathcal{M}_2$. Therefore, a related *bidirectional model transformation* where also $trans^{-1}$ can be directly computed might not be able to relate to each model of $\mathcal{M}_2$ a model in $\mathcal{M}_1$ using $trans^{-1}$.

Possible reason for this asymmetry can be, for example, that the models in $\mathcal{M}_2$ are more detailed and can thus describe structures or behavior which cannot be represented in $\mathcal{M}_1$. E.g., an assembled program might very well contain a whole bunch of unstructured goto statements, while a good programming language does explicitly exclude them and supports only well-structured loop constructs.

Another problem is that $trans^{-1}$ is not necessarily a function. If, for example, two models $M_1 \in \mathcal{M}_1$ and $M_1' \in \mathcal{M}_1$ with $M_1 \neq M_1'$ exist with $trans(M_1) = trans(M_1')$, we cannot define a unique result for $trans^{-1}$ for $trans(M_1)$. Examples for this case are several high level program constructs which may result in the same assembler code. E.g., a while and for loop could result in exactly the same assembler representation.

If we assume that $trans$ is an injection, we could conclude that $trans^{-1}$ must be a function and we name this a *bijective bidirectional model transformation* for $\mathcal{M}_1$ and $\mathcal{M}_2^*$. Otherwise, we have a *surjective bidirectional model transformation* for $\mathcal{M}_1$ and $\mathcal{M}_2^*$ and $trans^{-1}$ is a function from $\mathcal{M}_2^* \rightarrow \wp(\mathcal{M}_1)$ to encode that there might be several valid backward transformations.

For the bidirectional incremental case, we in addition have for a given model $M_1 \in \mathcal{M}_1$, its transformation $M_2 = trans(M_1)$, and an arbitrary modification $mod_2 : \mathcal{M}_2 \rightarrow \mathcal{M}_2$, that an *incremental model transformation* would allow to derive a modification $mod_1 : \mathcal{M}_1 \rightarrow \mathcal{M}_1$ such that it holds: $mod_1(M_1) \in trans^{-1}(mod_2(M_2))$.

We have to further restrict this condition if $\mathcal{M}_2^* \neq \mathcal{M}_2$ such that it must only hold for $mod_2(M_2) \in \mathcal{M}_2^*$. Otherwise we have to conclude that mod_2 is an inconsistent modification. E.g., the assembler code has been modified in such a manner that a code structure resulted which could not be the result of any program of the given programming language.

For the addressed *incremental model synchronization*, we require a bijective, bidirectional, incremental model transformation. In a case where for the resulting target model M_2' holds that $M_2' \in \mathcal{M}_2 \setminus \mathcal{M}_2^*$, we have to reject the modification in order to keep both models consistent.

3.2 Incremental Transformations and Updates

In order to make our algorithm incremental we have to take the correspondence model into account. Due to the construction principle of the triple graph grammar rules, each rule has at least one correspondence node in its pre-condition which thus is a necessary prerequisite for the application of the rule and therefore, the rule can be only applied if the required correspondence node was already

created in a previous transformation step. Additionally, each successful application of a rule results in at least one additional correspondence node. Therefore, in our transformation algorithm, a directed edge from the required correspondence node to the created one is inserted each time a rule is successfully applied. We include this link in our derived graph rewriting rules. The additional link between the correspondence nodes reflects the dependency and the execution order of the rules which will be used to extend our algorithm to the incremental case.

This observation can be exploited by using the created correspondence node as a starting point for a local searching strategy which reduces the costs for the required pattern matching. With the additional links between the correspondence nodes the correspondence model can be interpreted as a directed acyclic graph (DAG). It is a graph rather than a tree due to the fact that rules are allowed to have more than one correspondence node as a precondition. The graph is acyclic since in a rule application, we never connect already existing correspondence nodes by a link.

The incremental transformation and update algorithm traverses the correspondence nodes of the DAG using breadth-first search. For each correspondence node the algorithm checks whether an inconsistent situation has occurred. This is done by retrieving the rule which has been applied in the transformation process to create the correspondence node and checking whether it still matches to the current situation.

In the case that the structure of the applied rule still holds and only an additional attribute constraint evaluates to false, it is sufficient to propagate the attribute value change in the current transformation direction.

When the rule cannot be matched anymore, e.g., due to the deletion of a model element, we have found indeed an inconsistency. In that case, the algorithm has to undo the applied transformation rule. This is achieved by deleting the correspondence node and all created elements and unmarking the remaining nodes that have been involved in the right-side of the production. This is the last step of the update. However, note that, by deleting the correspondence node the precondition for all successors of the deleted correspondence node will not hold anymore. As a consequence, this leads both to the deletion of the succeeding correspondence nodes and the nodes in the class diagram referenced by the deleted correspondence nodes.

In the last step of the incremental transformation the algorithm searches for unmatched model elements and transforms those elements according to the triple graph grammar specification. The presented incremental algorithm can be used for unidirectional model transformations as well as for bidirectional model transformation and synchronization enabling round-trip engineering between models. We can further optimize our incremental algorithm if the involved models support change notifications. In that case, the presented transformation algorithm starts to traverse the DAG at the correspondence node connected to the modified element and not at the root of the DAG.

4 Evaluation

To evaluate our incremental approach, we use the model transformation example introduced earlier which synchronizes SDL block diagrams with a related UML class diagrams. In order to be able to evaluate large models with different characteristics, we wrote a parameterized synthesis algorithm for a hierarchical SDL block models where we can adjust the number of SDL blocks and the number of subblocks for each block. Therefore, we can in fact control the resulting out-degree w.r.t. rule dependencies in the resulting correspondence graph G_c by simply adjusting the number of subblocks for each block.

We further restrict our considerations to the forward direction as the backward case employs the same execution engine. The measurements have been done on an computer with an Intel(R) Pentium(R) m Processor with 1.80 GHz and 1,0 GB RAM. The complied rules and the execution engine have been run on Java 1.4.2_07 on top of the Windows XP Professional operating system.

4.1 Measured Synchronization Times

Taking the directed acyclic graph structure of the correspondence graph and the existence of a unique root node and leaf nodes into account, we can further assign to each node in the correspondence graph G_c the related height which relates to the length of the longest path from that node to a node without successor (leaf). This height can then also be related to the connected graph nodes of the source and target model. The height of the model is further simply the height of the root node.

While for the batch-oriented processing the required model synchronization efforts are the same for every modification in the source model, in the incremental case the specific effort required for a specific small modification on the source model depends on the height of the related correspondence nodes. We thus also characterize small modifications by the related height. This dependency is that the larger the related height is the higher are the efforts for the required processing. One extreme case is the model root. The required computation of the incremental transformation in fact involves the whole model.

Another factor which is relevant here is of course the out degree of the correspondence nodes. Obviously, a higher out-degree results in a higher computation effort for the same height as more subordinated nodes have to be subject to the application of the transformation rules. On the other hand a higher out-degree results in a much smaller height of the model and much more nodes with smaller height.

The resulting measurements for a SDL model with 5.000 blocks for the batch and incremental algorithm are depicted in Figure 4. On the x-axis the different possible heights of the small modifications are enumerated and the related measurement results for the models with different out-degrees (Out n) are provided. In addition, we added the results for the batch-oriented algorithm (Out n - batch) which are independent of the height and thus are simply straight lines.

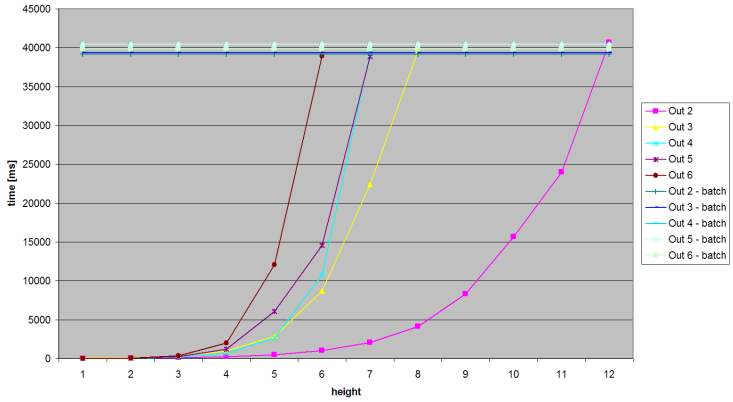

Fig. 4. Efforts for the synchronization after a modification

The expected effect which can be observed is that for larger out degrees we have smaller maximal height and thus the required efforts increase more rapidly with increasing height. For all cases holds that in case of the maximal height the same effort as for the batch processing can be observed.

4.2 Average Synchronization Costs

To derive a useful performance prediction from these measurements, we will further combine them to derive reasonable estimates for the average case of modifications.

Depending on the average height of the related correspondence nodes involved in the modification, a reasonable speedup w.r.t. a batch processing of the whole transformation can be observed. To relate this observation to a reasonable estimation of the average performance, we derive an average case effort estimation starting with the assumption that all changes have the same likelihood. For n the number of correspondence nodes, $h_{\max}$ the maximal height of the correspondence graph, n_h the number of correspondence nodes with height h, and T_h the measured time for processing a small modification in ms, the mean value for the time T_a required for the processing of an arbitrary small modification is then: $T_a = (\sum_{h=0}^{h_{\max}} n_h T_h)/n$.

Using the data about n, $h_{\max}$, n_h, and T_h presented in Figure 4 we thus have the average computation times as reported in Figure 5. In addition, we also computed the values for smaller models.

We can observe that with increasing out-degree the average synchronization time is reasonable small (about 20-30 ms) and increases only minimally with the model size (1-3 ms). For smaller out-degree the average case becomes more costly (60-90 ms) and also increases significantly (100-180 ms). The visible steps in the calculated average times and the following smooth decrease in the average time are related to an increase in height and the related result that the lower ranks of the correspondence DAG are that well balanced.

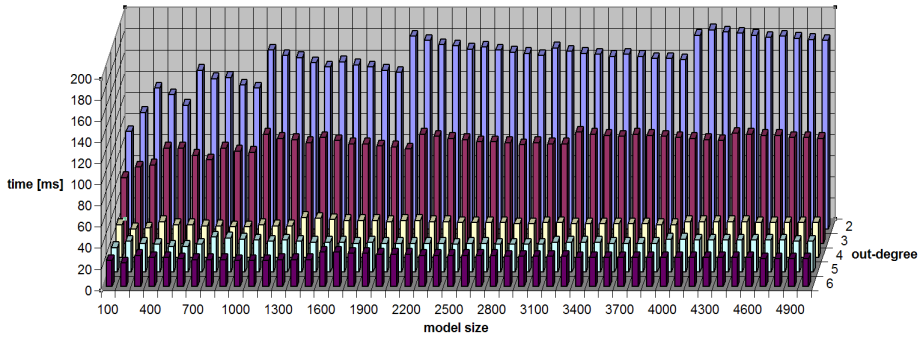

Fig. 5. Average computation times for small modifications and different out degrees

4.3 Discussion

Taking the directed acyclic graph structure of the correspondence graph into account, we know that only the nodes of the correspondence graph G_c beneath the correspondence nodes which are directly related to the modified node or edge have to be recomputed. While this already restricts the required computation effort, in the worst case clearly nearly the same effort as in the non incremental case is required.

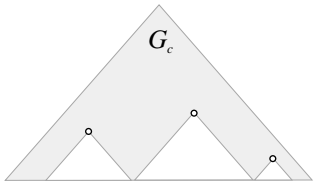

Fig. 6. Incremental application of the TGG rules for arbitrary modifications

In Fig. 6, the resulting effect on the acyclic directed correspondence graph is depicted. For sake of visual presentation, we use a tree rather than a DAG. Depending on the average height of the correspondence nodes in the tree/graph involved in the modification, a reasonable speedup w.r.t. a batch processing of the whole transformation can be expected.

The described observation for the average case can be backed up by the following theoretical derivation of the complexity: For n nodes and a maximal depth $d_{\max}$ we roughly have $n \approx \exp(d_{\max})$ nodes in a tree. If we further assume that there are about $N(d) \approx \exp(d)$ nodes for a specific depth d and that the number of rule applications T for processing an update for a correspondence node with depth d is $T(d) \approx \exp(d_{\max} - d)$, the mean number of rule applications T_m for processing an update for an arbitrary correspondence node assuming an

average distribution is:

$$T_m \approx \frac{\sum_{d=0}^{d_{\max}-1} N(d) * T(d)}{n} \approx \frac{\sum_{0=0}^{d_{\max}-1} \exp(d) * \exp(d_{\max} - d)}{\exp(d_{\max})}$$
$$\approx \frac{\sum_{d=0}^{d_{\max}-1} \exp(d_{\max})}{\exp(d_{\max})} = \frac{(d_{\max} - 1) * \exp(d_{\max})}{\exp(d_{\max})} = (d_{\max} - 1)$$

In contrast to repeat the full computation of the correspondence graph which would require $n \approx \exp(d_{\max})$ rule applications for a model with n nodes, we only require $(d_{\max} - 1)$ rule applications in the average case. Thus as $d_{\max} \approx \log(n)$, we have a *effectively incremental* solution as the impact of the model size in the average case is only in $O(\log(n))$ and not $O(n)$ as for batch processing.

It is to be noted that models in practice often have 7 or more elements at the same abstraction level which are then further refined by assigning submodels to each element, while we have looked into out-degrees from 2 to 6. The considered data indicates that for higher out-degrees we can expect even better performance than for the smaller out-degrees and thus the considered cases are from a practical point of view the worst cases.[2]

5 Related Work

Motivated by the Model-Driven Architecture (MDA) [1] and OMG's Request for Proposal (RFP) on Query/Views/ Transformations (QVT) [8], model transformation has been put into the focus of many research activities. Meanwhile, a first version of the Final Adopted Specification [9] is published and the final version is expected in the course of this year. In this specification, incremental model transformations are an important issue. However, to the best knowledge of the authors, up to now there is no publicly and freely available tool implementing the QVT standard with incremental updates for model synchronization.

A tool supporting incremental model transformations is the *Model Transformation Framework* (MTF) [10] developed by IBM. Unfortunately, so far, there is no performance data nor any publication describing the used approach available.

Nevertheless, the RFP has lead to a large number of approaches for model transformation - each for a special purpose and within a particular domain with its own requirements [11]. A class of transformation approaches comprises graphical transformation languages which are based on the theoretical work on graph grammars and graph transformations. These approaches interpret the models as graphs and the transformation is executed by searching a pattern in the graph and applying an action which transforms the pattern to a new data structure. However, these languages do not provide any explicit traceability information about the model transformation. This prevents both incremental transformations

[2] The absolute worst case is a linear list where the effort is of course proportional to the height. We are, however, not aware of any example where the metamodels and their model instances in practice result in a linear list.

and consistency maintaining activities for model synchronization after an applied transformation. Additionally, in the most graph grammar based approaches, the transformation must be specified for each transformation direction separately. Hence, they are not well suited for the specification of bidirectional model transformation and synchronization.

In contrast to that, triple graph grammars are a special technique for the specification and execution of bidirectional transformations. Triple graph grammars were motivated by integration problems between different tools where interrelated documents have to be kept consistent with each other [12–14]. In this field, triple graph grammars are used for the maintenance of the required traceability links between different document artifacts. In [13] the transformation algorithm operates interactively and incrementally. In contrast to our approach, the transformation algorithm relies on the type of so called *dominant increments*. The incremental transformation approach in [15] is triggered by user actions like creating, editing, or deleting elements. A complete model transformation from scratch is not in the focus of the approach whereas our approach handles both cases. However, some of the work served as a starting point for our approach. In particular, we rely on the proposed attribute update propagation techniques [13, 14] and the correspondence dependency introduced by [12].

6 Conclusion and Future Work

We have presented our approach for the efficient and incremental model synchronization with the model transformation approach triple graph grammars. Our solution at first is visual, formal, and bidirectional which are all characteristics it inherits from triple graph grammars. In addition, our extension of the rule execution facilitates an incremental application which takes the acyclic dependencies present in the correspondence graph into account and therefore results in an effectively incremental solution for the model synchronization problem.

We have realized our approach in the Fujaba Tool Suite[3]. The available tool support includes the visual specification of the triple graph grammar rules, the automatic extraction of the resulting graph rewriting rules, and an execution engine for the incremental execution of these rules.

The paper provides measurements for the effort required for an example model transformation task and the related model synchronization in particular for the case of *large* models. To our knowledge, similar data is currently not provided by any related approach. We hope that this will change in the future and that this contribution is a first step towards setting up benchmarks for model transformation such that the finding can be based on commonly agreed examples and that the different approaches can be systematically compared.

As future work we plan to provide a QVT compatible front-end for our approach which maps the QVT semantics to triple graph grammars in order to make the technology available to a broader audience.

[3] www.fujaba.de

References

1. OMG: MDA Guide Version 1.0.1. (2003) Document – omg/03-06-01.
2. Wagner, R., Giese, H., Nickel, U.: A Plug-In for Flexible and Incremental Consistency Management. In: Proceedings of the Workshop on Consistency Problems in UML-based Software Development II (UML 2003, Workshop 7), Blekinge Institute of Technology (2003) 78–85
3. Gardner, T., Griffin, C., Koehler, J., Hauser, R.: Review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations towards final Standard. OMG, 250 First Avenue, Needham, MA 02494, USA. (2003)
4. Schürr, A.: Specification of graph translators with triple graph grammars. In Mayr, E.W., Schmidt, G., Tinhofer, G., eds.: Graph-Theoretic Concepts in Computer Science, 20^{th} International Workshop, WG '94. Volume 903 of LNCS., Herrsching, Germany (1994) 151–163
5. Schäfer, W., Wagner, R., Gausemeier, J., Eckes, R.: An Engineer's Workstation to support Integrated Development of Flexible Production Control Systems. In Ehrig, H., Damm, W., Desel, J., Groše-Rhode, M., Reif, W., Schnieder, E., Westkämper, E., eds.: Integration of Software Specification Techniques for Applications in Engineering. Volume 3147 of Lecture Notes in Computer Science (LNCS). Springer Verlag (2004) 48–68
6. International Telecommunication Union (ITU), Geneva: ITU-T Recommendation Z.100: Specification and Description Language (SDL). (1994 + Addendum 1996)
7. OMG 250 First Avenue, Needham, MA 02494, USA: (Unified Modeling Language Specification Version 1.5)
8. OMG: OMG/RFP/QVT MOF 2.0 Query/Views/Transformations RFP. http://www.omg.org/mda/. (2003)
9. OMG: MOF QVT Final Adopted Specification, OMG Document ptc/05-11-01. (http://www.omg.org/)
10. Griffin, C.: Eclipse Model Transformatioin Framework (MTF), available at http://www.alphaworks.ibm.com/tech/mtf. IBM. (2006)
11. Czarnecki, K., Helsen, S.: Classification of Model Transformation Approaches. In: Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, USA, 2003. (2003)
12. Lefering, M., Schürr, A.: Specification of Integration Tools. In Nagl, M., ed.: Building Thightly-Integrated (Software) Development Environments: The IPSEN Approach. Volume 1170 of Lecture Notes in Computer Science., Springer Verlag (1996) 324–334
13. Becker, S., Lohmann, S., Westfechtel, B.: Rule Execution in Graph-Based Incremental Interactive Integration Tools. In: Proc. Intl. Conf. on Graph Transformations (ICGT 2004). Volume 3256 of LNCS. (2004) 22–38
14. Königs, A., Schürr, A.: Tool Integration with Triple Graph Grammars - A Survey. In Heckel, R., ed.: Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques. Volume 148 of Electronic Notes in Theoretical Computer Science., Amsterdam, Elsevier Science Publ. (2006) 113–150
15. Guerra, E., de Lara, J.: Event-Driven Grammars: Towards the Integration of Meta-Modelling and Graph Transformation. In: International Conference on Graph Transformation (ICGT'2004). Volume 3265 of LNCS. (2004) 54–69