Preventing Information Loss in Incremental Model Synchronization by Reusing Elements*

Joel Greenyer[1**], Sebastian Pook[2], and Jan Rieke[1**]

[1] Software Engineering Group, Heinz Nixdorf Institute
Department of Computer Science
University of Paderborn, 33098 Paderborn, Germany
{jgreen|jrieke}@uni-paderborn.de

[2] Heinz Nixdorf Institute
University of Paderborn, 33098 Paderborn, Germany
Sebastian.Pook@hni.uni-paderborn.de

Abstract. The development of complex mechatronic systems requires the close collaboration of multiple engineering disciplines. Hence, multidisciplinary system engineering approaches have been developed. However, the refinement of discipline-specific aspects of the system, for example the implementation of software controllers, still requires discipline-specific models and tools. During the development, changes in these discipline-specific models may affect other disciplines' models. Thus, inconsistencies are likely to occur, leading to increased development time and costs if they remain undetected. Bidirectional model synchronization techniques aim at automatically resolving such inconsistencies. Existing synchronization algorithms today, however, fail in this application scenario, because synchronization steps often unnecessarily destroy and re-create elements, which damages model parts that are not subject to the synchronization. In order to solve these issues, we present a novel synchronization technique based on Triple Graph Grammars with improvements regarding the reuse of model elements.

Keywords: Incremental Model Synchronization, Mechatronic System Design, Triple Graph Grammars (TGG), Information Retainment in the Target

1 Introduction

The development of mechatronic systems, from modern household aids to transportation systems, requires the close collaboration of multiple disciplines, such as mechanical engineering, electrical engineering, control engineering, and software engineering. Usually, a discipline-spanning *system model* is created first. Next,

* This work was developed in the course of the Collaborative Research Center 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, funded by the Deutsche Forschungsgemeinschaft.

** supported by the International Graduate School Dynamic Intelligent Systems.

engineers from each discipline develop *discipline-specific models* in parallel, using different modeling languages and tools. As changes to these models are likely to affect other disciplines, avoiding inconsistencies is crucial.

To automatically synchronize the different models used during the development, a concept is needed to bidirectionally propagate changes between the different models: if, for instance, the discipline-spanning system model is changed, these changes must be propagated from this (*source*) model to the discipline-specific (*target*) models. Bidirectional model transformation techniques are a promising approach for such synchronization scenarios. However, existing synchronization algorithms [4,5,18,8] are not sufficient for such a scenario: When changes to a source model are propagated, often too many elements of the target models are unnecessarily deleted and recreated. This severely damages parts of the target model which are not subject to the transformation, but referenced the deleted elements. Such synchronization issues arise in many model-based development scenarios: different models are created for different purposes, and overlap in the information they contain, e.g., models for specification and models for testing. We present an improved synchronization algorithm based on Triple Graph Grammars (TGGs) [14], a rule-based formalism for declaratively specifying relations between models. This algorithm prevents unnecessary deletions by providing flexible repair operations.

The paper is structured as follows. Sec. 2 describes the development of mechatronic systems and introduces the example. In Sec. 3, we give a short introduction to TGGs and model synchronization approaches. The main contribution, our improved synchronization algorithm, is described in detail in Sec. 4. Finally, we summarize related work in Sec. 5 and conclude the paper in Sec. 6.

2 Development of Mechatronic Systems

Design guidelines for mechatronic systems, like VDI 2206 [17], or development methods elaborated in the Collaborative Research Center (CRC) 614 "Self-Optimizing Concepts and Structures in Mechanical Engineering" in Paderborn, propose that experts from all disciplines collaborate in a first development phase, called the *conceptual design*. Together they work out the *principle solution*, a system model that captures all interdisciplinary concerns. A core part of this interdisciplinary system model is the *active structure*, which shows how the system is composed of different system elements, how they are hierarchically structured, and how they affect each other by flows (e.g., information or energy flows).

The principle solution then serves as a basis for the discipline-specific *refinement* phase. However, the principle solution rarely captures all interdisciplinary concerns and, therefore, cross-disciplinary changes may become necessary during the discipline-specific refinement phase. These changes then have to be propagated among the discipline-specific models. This is realized by first updating the interdisciplinary system model with the information relevant to other disciplines and then propagating these changes to affected discipline-specific models.

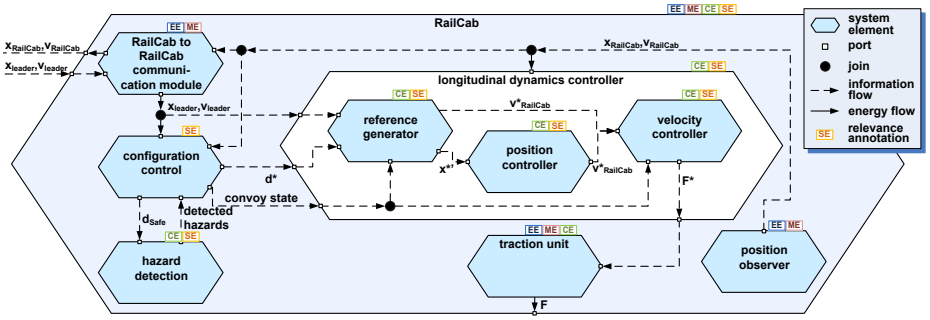

Fig. 1. Parts of the active structure of the RailCab system

Gausemeier et al. [3] described such a process from a methodological viewpoint, showing the applicability of model transformation techniques in general.

As an example, we consider the *RailCab* project[3]. Its vision is that, in the future, the schedule-based railway traffic will be replaced by small, autonomous RailCabs, which transport passengers and goods on demand, being more energy efficient by dynamically forming convoys. Fig. 1 shows parts of the active structure of a RailCab. This diagram kind is part of an interdisciplinary specification language [2], which we call the *Mechatronic Modeling Language* (MML). We consider the refinement in the discipline of software engineering using *Mechatronic UML*. Mechatronic UML is a modeling language for the development of distributed, safety-critical real-time systems, especially to model the software architecture and the behavior of the system and its components [1]. It allows us to specify *hybrid components*, which include both discrete and continuous behavior, and dynamic *reconfigurations* of components.

Let us take a closer look at the active structure in Fig 1. The longitudinal dynamics controller is responsible for controlling the traction unit. The control strategy for the velocity is reconfigured based on the current convoy state: Usually, a setpoint value for the speed, $v^*_{RailCab}$, is used, calculated by the reference generator. In convoy mode, the controllers are reconfigured so that instead the position controller becomes active and the velocity is controlled based on the distance to the leading RailCab. The basic reconfiguration behavior is described in the principle solution, but the details are implemented during the discipline-specific refinement. For details, we refer to Gausemeier et al. [3]. The small *relevance annotations* at the top of each system element mark which system element is relevant to which discipline (e.g., "SE" denotes Software Engineering). Thus, these annotations define discipline-specific views on the active structure.

Inconsistencies may easily arise during the development. Consider the following process as an example.

1. The discipline-specific models are generated from the principle solution by different *initial* model transformations. Fig. 2 shows how different elements of the active structure correspond to Mechatronic UML model elements.

[3] Neue Bahntechnik Paderborn/RailCab: `http://www-nbp.uni-paderborn.de/`

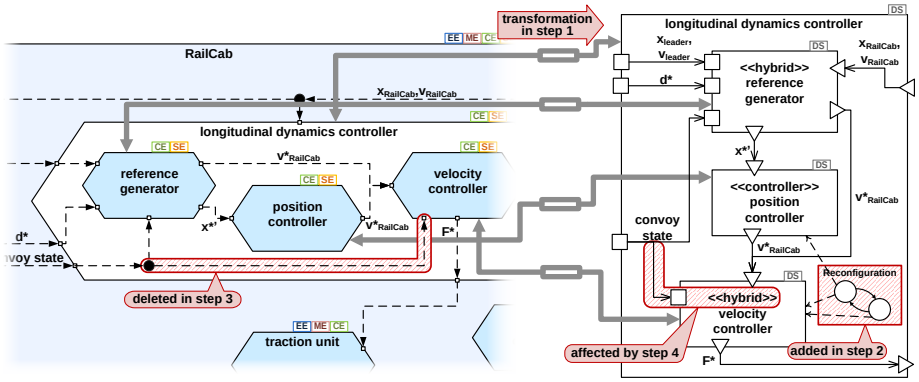

Fig. 2. Transition from the active structure to a Mechatronic UML component diagram

2. The disciplines' engineers start refining their models. E.g., the software engineer defines behavior for components; especially he elaborates on the reconfiguration behavior for the longitudinal dynamics controller. This discipline-specific information is typically not reflected back to the principle solution.
3. A flaw is identified within the original principle solution: the velocity controller inside the longitudinal dynamics controller receives information about the current convoy state, but this is unwanted and possibly misleading for the engineers implementing it. So the principle solution is changed, and this information flow and its ports are removed.
4. After the only discrete port of the velocity controller has been removed, the corresponding component in Mechatronic UML must not be a hybrid component anymore, but a simple controller component. (Controller components, in contrast to hybrid components, are not further refined by the software engineers, but entirely implemented by control engineering. Thus, it is just a placeholder for controller code.) Therefore, the model synchronization updates the software model and changes the hybrid component to a controller.

The main challenge in step 4 is that the model synchronization has to replace some parts of the Mechatronic UML model, *but in a way that the discipline-specific information introduced in step 2 is not destroyed or becomes invalid.* Here, the reconfiguration chart developed in step 2 references the hybrid component velocity controller. Deleting this hybrid component and recreating it as a controller component would *damage this reconfiguration chart*, because the chart would still reference the deleted hybrid component.

3 Triple Graph Grammars

Bidirectional model transformation techniques are a promising approach for automatically synchronizing the different models during the development. Here, we use a concept called *Triple Graph Grammars* (TGGs) [14]. TGGs are a rule-based formalism that allows us to specify how corresponding graphs or models

can be produced "in parallel" by linking together two graph grammar rules from two different graph grammars. More specifically, a TGG rule is formed by inserting a third graph grammar rule to produce the so-called *correspondence* graph that links the nodes of the other two graphs. TGGs can be interpreted for different transformation and synchronization scenarios. Before we describe these scenarios, let us consider the structure of TGG rules.

3.1 Triple Graph Grammar Rules

Fig. 3a illustrates a TGG rule, SystemElementToHybridComponent, which is taken from a TGG that defines the mapping between MML and Mechatronic UML.

TGG rules are non-deleting graph grammar rules that have a left-hand side (lhs) and a right-hand side (rhs) graph pattern. The nodes appearing on the lhs and the rhs are called *context nodes*, displayed by white boxes. The nodes appearing on the rhs only are called *produced nodes*, displayed by green boxes, labeled by "++". Accordingly, there are *context edges*, displayed by black arrows, and *produced edges*, displayed by green arrows and "++" labels.

In TGGs, graphs are *typed* and *attributed*. When working with models and meta-models in terms of MOF [11], this means that the host or *instance* model contains objects and links that are instances of classes and references of a given meta-model. Accordingly, the nodes and edges in the rules are typed over the classes and references in a meta-model. Nodes are labeled in the form "*Name*:*Type*". For instance, the nodes in the left column of rule SystemElement-ToHybridComponent are typed by the classes Package and SystemElement from the MML meta-model. The edge is typed over the reference packagedElement.

The columns of a TGG rule describe model patterns of different meta-models and are called *domains*. The left-column production states that when there is a package in MML, we can add a system element and a link between them. The right column of the rule represents the graph grammar production for creating components in packages in Mechatronic UML. In the middle, there is the production of the correspondence structure between the models.

Our TGG rules further introduce the concept of *attribute constraints* and *application conditions* (depicted by yellow, rounded rectangles in Fig. 3). Attribute constraints are attached to nodes and have expressions of the form

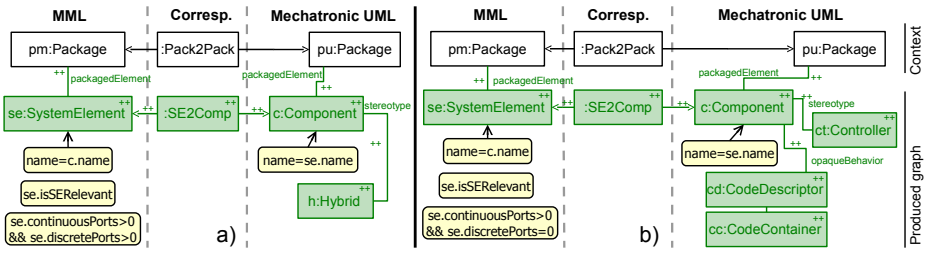

Fig. 3. a) SystemElementToHybridComponent, b) SystemElementToController

$\langle prop \rangle = \langle expr \rangle$, where $\langle prop \rangle$ is a property of the node's type class, and $\langle expr \rangle$ is an OCL expression that must conform to the type of $\langle prop \rangle$. Node names can be used as variables in the OCL expression. Attribute constraints constrain the attribute value of an object. E.g., rule SystemElementToHybridComponent has two constraints that express that the name of the component has to be equal to the name of the opposite component. A rule application is not valid if the attributes do not equal the specified values. Application conditions are OCL expressions over model properties that evaluate to a Boolean value. A rule application is not valid if an application condition evaluates to false. E.g., the constraints on the left of the rule SystemElementToHybridComponent restrict the applicability to cases where there are both continuous and discrete ports at the system element and where the component is relevant to software engineering.

To express the relation between MML and Mechatronic UML, we defined a set of TGG rules. Rule SystemElementToHybridComponent defines how system elements with both discrete and continuous information flow ports correspond to hybrid components. Rule SystemElementToController (Fig. 3b) relates system elements without discrete but with continuous ports to controller components.

A TGG may only refer to a subset of the classes defined in the domain metamodels. Instances of other classes are allowed to occur in the models. We then say that they are not *subject to the transformation*. For instance, if reconfiguration behavior is modeled in Mechatronic UML, it is not subject to the transformation.

3.2 Application Scenarios

We can interpret TGGs for different *application scenarios*. One scenario, called *forward transformation*, is to create one "target" graph corresponding to a given "source" graph. In this case, a TGG rule is interpreted as follows: First, the context pattern of the rule is matched to *bound* model elements, which are objects and links that were previously matched by another rule application. Second, the source produced pattern is matched to yet *unbound* parts in the source model. If a matching respecting these conditions is found, the produced target and correspondence patterns are created and bindings are created for the newly matched and created elements.[4] The *backward* direction works accordingly, reversing the notion of source and target. We refer to Greenyer and Kindler [6] for further details on TGGs and the binding semantics.

Our TGG engine interprets attribute constraints (of the form $\langle prop \rangle = \langle expr \rangle$) as assignments in the target domain. If a TGG rule shall support both transformations directions, assignments must be specified for both directions.

3.3 Incremental Model Synchronization

In a situation where a triple of corresponding models is given and a change occurs in one domain model, this change can be propagated by *incrementally*

[4] In the following, we use the term *binding* when referring to a single node-to-object or edge-to-link match, and *matching* for a set of those bindings (i.e., when a whole pattern is matched to several elements).

updating only the affected parts of the model. Algorithms for this problem have been described before [5,4,18,8]. In the following, we explain the shortcomings of existing incremental synchronization approaches.

Giese and Wagner suggest the following approach [5]. When a change occurs in the source model, the rule application(s) which are violated by the change (i.e., there is no longer a valid matching of the rule) are revoked. Second, all rule applications that depend on the revoked rule(s) are also revoked. During that process, the target model elements created by the revoked rule applications are destroyed, and the source model elements become unbound. Finally, the transformation is re-run for the unbound source model elements.

Giese and Hildebrandt present another algorithm [4]. When a change occurs in the source model, it tries to repair the rule applications that are violated by the change. For instance, if the change is a move of an element on the source side, a rule application can be repaired by changing the link from the corresponding (target-side) element to its old (target-side) parent such that it is now linked to the new corresponding parent on the target side. These repairs are performed by pre-generated repair operations that are derived from the rules. Only if such a repair operation is not possible, the rule application is revoked. Then the algorithm tries to apply another rule to these unbound source model elements. However, these repair operations are only able to modify links; whenever it is not possible to repair a rule application by changing a link, the rule must be revoked and the target model elements are destroyed. In the following, we present an example where this approach leads to the unwanted loss of information in the target model. Our improved algorithm resolves this problem.

4 Improved Synchronization

As the deletion of elements should be prevented if possible, it is reasonable not to immediately destroy elements when a rule application is revoked. Thus, these elements are just *marked for deletion* by our improved synchronization. The novelty of our algorithm is that it can reuse these removed objects and links during later rule application by explicitly *searching for matches in the set of elements marked for deletion*. Only if such a deletion-marked element cannot be reused by a new rule application, it will be ultimately destroyed. The problem is that there may be several ways of how these elements can be reused. A particular challenge is then to determine the "best" way to reuse these elements.

Next, we present an example and overview the improved synchronization algorithm. Then we give an extended example with different ways of reusing elements and we discuss heuristics to determine which reusable pattern may be best. Finally, we discuss the details of the *partially reusable pattern search* and conclude with a runtime evaluation.

4.1 Improved Synchronization Example

We assume that our models are in a consistent state, e.g., after an initial transformation as shown in Fig. 2. In particular, all elements are bound by a TGG

rule application. Then, as described in Sec. 2, the software engineer adds a reconfiguration chart to the Mechatronic UML model. Next, the information flow convoy state to the system element velocity controller is deleted from the MML model, and with it its corresponding port (crossed-out in Fig. 4a). The rule ap-

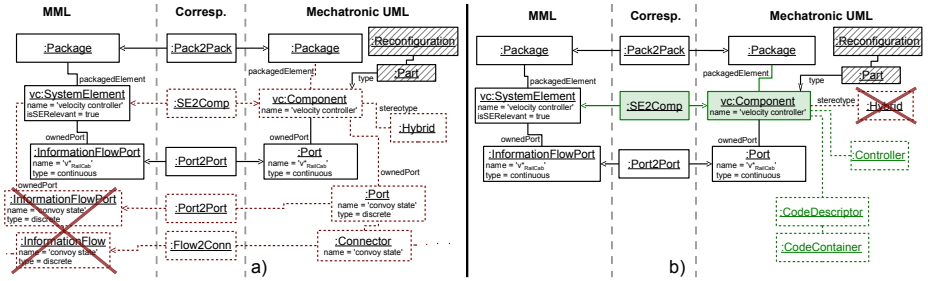

Fig. 4. Abstract syntax after rule revocation, a) after deleting a flow and the revocation of rules, b) after applying the new rule (with reusing elements by repair operations)

plications that previously translated the information flow and the information flow port are now structurally invalid[5]. Therefore, these rule applications are revoked, which means removing all its bindings and marking the correspondence and target produced objects and links as deleted (denoted by dashed lines in Fig. 4a). Additionally, the application of SystemElementToHybridComponent that mapped the velocity controller system element to a hybrid component becomes invalid because the constraint `continuousPorts>0&&discretePorts>0` is violated, as there is no discrete port any more. So, this rule application has to be revoked by marking its produced part as deleted, too.

We now try to apply new rules immediately, instead of first revoking depended rule applications. Here, rule SystemElementToController (Fig. 3b) can be applied. Its context is matched onto the package objects in MML and UML and the correspondence :Pack2Pack. Updating incrementally in forward direction, the produced source (MML) pattern (consisting only of se:SystemElement) is searched. It matches the velocity controller system element, which is now unbound due to the revocation of SystemElementToHybridComponent. Also the constraints hold, as they require no discrete port. A normal rule application would simply create the correspondence and target patterns. Instead, our improved synchronization first searches for a pattern matching in the set of elements marked for deletion. Two elements in this set can be reused: starting the search from the velocity controller system element, our algorithm finds and reuses the (deletion-marked) :SE2Comp correspondence and the velocity controller component.

No other previously deleted element can be reused by this rule. Unfortunately the matching is not complete yet, as there are no existing objects to match the

[5] We refrain from showing these TGG rules, as they simply map one information flow (resp. one information port) to one connector (resp. one port).

Controller, CodeDescriptor, and CodeContainer nodes of rule SystemElementToController. The algorithm uses this *partial pattern matching* anyway. Now we can apply the rule as follows. First, remove the "deleted" flag from everything that has been reused and binding these elements again. Second, because the match of the TGG rule is not yet complete, additional links and objects are created as required for this rule. Unfitting links of single-valued references are moved. This process is called *repair operation*.

Fig. 4b shows the situation after the rule application. The algorithm reused the :SE2Comp correspondence and the velocity controller component (shaded in Fig. 4b). It created new instances of Controller, CodeDescriptor, and CodeContainer, and set the appropriate links (dashed in Fig. 4b), as no reusable element could be found for them. The Hybrid stereotype object could not be reused. Thus, it is ultimately destroyed (crossed-out in Fig. 4b). Also the convoy state port, the connector and their correspondences are destroyed (not shown in Fig. 4b).

By reusing the velocity controller component, which was previously marked for deletion, a dangling edge from the model-specific reconfiguration specification (hatched in Fig. 4a and 4b) is prevented. Additionally, any further model-specific information that is attached to the component is also preserved. Furthermore, a deletion and recreation of the component would have rendered the context for the PortToPort rule invalid. Thus, the old synchronization algorithm would have to revoke this and possibly further rule application.

4.2 Improved Synchronization Algorithm

In summary, our improved synchronization algorithm works as follows:

1. Iterate over all TGG rule applications in the order they were applied, and if the application has become invalid due to changes in the source model
   a. Remove the bindings of the produced graph, and mark the correspondence and target produced elements previously bound to this graph as deleted.
   b. If the same or other rules are applicable in the forward direction, i.e., the context and the source produced graph match,
      i. search for a pattern of elements marked for deletion that "best" matches part of the rule's correspondence/target graph structure
      ii. apply the rule by reusing this pattern, creating the remaining correspondence/target pattern, and enforcing attribute value constraints.

   Continue checking the next rule application or terminate if all applications have been checked.
2. Finally, effectively destroy elements that are still marked for deletion.

The concept to "mark for deletion" allows us to remember the elements that might be reusable. It is the basis for intelligently reusing model elements during the synchronization, which is the main improvement over previous approaches.

In the example above there was only one possible partially reusable pattern, but often there are several partial matchings that reuse more or less elements. In fact, some partial matchings may reuse elements in an unintended way. Therefore, we first calculate all possible partial matchings, and then choose the most

reasonable. In the following, we present an extended example in which two reuse possibilities occur. Next, we discuss the implementation details of the partially reusable pattern search, which computes the different reuse options (step 1b-i).

4.3 Selection of Elements to be Reused

The heuristic for the "best" partial matching is generally to take the partial matching that reuses most elements. However, also considering existing correspondence information can be vital, as we show in the following example. Let us assume that we start with a consistent state, as in the previous example. First, as before, the discrete port and its information flow are deleted from the MML velocity controller. Second, for some reason the position controller's relevance flag for the software engineering discipline is removed. Therefore the application of rule SystemElementToHybridComponent for the velocity controller and the application of rule SystemElementToController for the position controller is revoked. The result is shown in Fig. 5, with user deletions crossed-out and deletion-marked elements depicted by dashed lines.

As explained before, rule SystemElementToController is now applicable at the MML velocity controller. In addition to the partial pattern match described before (see Fig. 4b), there is a second promising partial match now (marked with $\surd$s in Fig. 5). It is found by the partial pattern matching search starting from the context :Package object of Mechatronic UML. This partial match reuses the deleted position controller component, its controller, code descriptor and code container. However, the existing correspondence node does not fit (marked with a ⊛): It points to the MML position controller, but it must be connected to the MML velocity controller (because the node se:SystemElement is already matched to the velocity controller). Additionally, the attribute constraint must be repaired, changing the component's name to "velocity controller" (also marked with ⊛).

Although this alternative partial matching reuses more elements than the partial matching used in Fig. 4b, it is in fact an example where the reuse is unintended, because it creates a correspondence between the "wrong" elements. At first glance, this may not seem to be a problem, because the change propagation will adapt all links and attribute values in the target model to satisfy the constraints posed by the TGG. However, there may be elements in the target

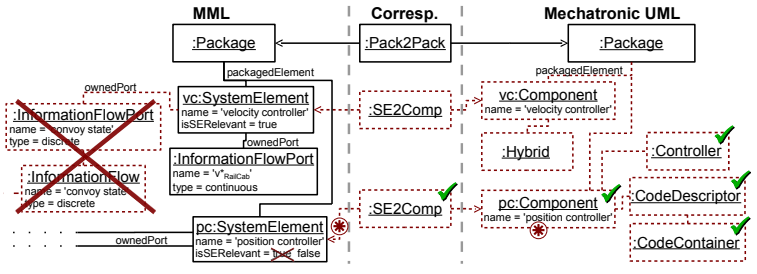

Fig. 5. "Wrong" partial pattern match

model that are not subject to the transformation which reference these reused elements. A "wrong" reuse means that these elements now reference completely altered objects that have changed in meaning. Therefore, it is reasonable to favor such partial matchings where a correspondence node is reused without changing its correspondence links. In this way, only previously corresponding elements are reused, typically resulting in the intended reuse of elements.

Note that in terms of the TGG semantics it is not relevant in which way existing elements are reused (or whether they are reused at all). A particular reuse of elements may only be more or less harmful to the elements that are not subject to the transformation. Our synchronization algorithm only produces triples of models where the elements that are subject to the transformation form a valid triple model according to the TGG. That is because (a) when a rule is applied, reused objects and links will be modified so that they fit the rules, and (b) at the end of a synchronization run, unused objects that are marked for deletion will be actually destroyed. Therefore, after a successful synchronization, every rule holds and no remainders of revoked rules exist. Thus, arguing informally, fundamental TGG transformation properties like termination, correctness or completeness should be unaffected by our new algorithm. However, a formal discussion of these properties is outside of the scope of this paper.

4.4 Partial Reusable Pattern Matching Algorithm

In the following, we describe the data structure we use for the partially reusable pattern search and discuss how the algorithm searches for partial matchings.

All possible partial matchings are computed by creating a tree structure. The root of this tree is a vertex which represents the matching of the context and the source produced domain pattern (computed in step 1b). Each edge of the tree represents a step of the pattern matching which binds a new node. Each other vertex is labeled with a single node binding (a node-object tuple). Additionally, it is labeled with its pattern matching *depth*, which is the depth in the recursion of a depth-first pattern matching algorithm. Each vertex of the tree therefore represents a (partial) matching of the rule, recursively defined by the node binding of the vertex and those of its parent.

The resulting tree reflects the pattern matching search: When traversing the rule, the algorithm adds a new child vertex for each successful pattern matching step (i.e., whenever it finds a new candidate object for a node). Thus, a vertex has more than one child when there are different possibilities to match a node.

Fig. 6 shows a part of the matching tree that is the result of a partial pattern matching search of rule SystemElementToController (Fig. 3b) in the set of deleted elements from Fig. 5. As described, the root of this tree contains the matching of the context and the source produced domain pattern. The different bindings of this matching are shown in the form "*Node*:*NodeType* → *Object*:*ObjectType*", where *Node* represent a node from the TGG rule and *Object* is the matched object. The algorithm starts a search from every binding in the root. Let us assume it first tries to match the TGG rule node pu:Package and finds the not yet bound outgoing edge packagedElement to c:Component (Fig. 3b). The

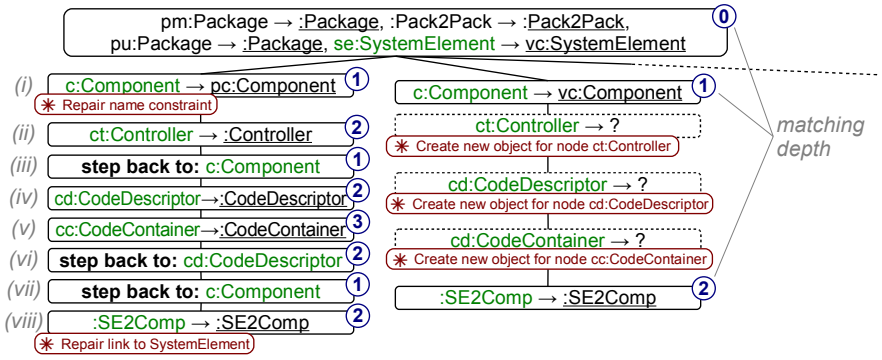

Fig. 6. Matching tree resulting from searching the produced pattern of SystemElement-ToController in the set of deleted elements from Fig. 5

algorithm now has two options on how to match this node: Both the objects position controller and the velocity controller components match. So a new vertex is created for both (the left one is marked with (i) in Fig. 6), each with $depth = 1$ (denoted as the circled number in the upper right corner of the vertices in Fig. 6).

The left subtree contains the "wrong" partial matching possibility (see Fig. 5): The algorithm continues with matching the ct:Controller node to the controller object and adding a vertex with $depth = 2$ for it (ii). Then, as there is no unbound node connected to the ct:Controller node, the search must be continued at the previous node, decreasing the *depth* (iii). Here, the previous node is simply the node of the parent vertex, c:Component. Next, the cd:CodeDescriptor (iv) and the cc:CodeContainer (v) is matched. Again, as no unbound node connected to cc:CodeContainer exists, the algorithm steps back in the pattern matching, i.e., returns to the previous node, cd:CodeDescriptor (vi). There is also no unbound node connected to cd:CodeDescriptor. At this point, the previous node is not the node of the parent vertex. Therefore, the previous node is identified using the *depth* counter: we walk up the tree and select the first vertex v with $v.\text{depth} < currentVertex.\text{depth}$, which is c:Component (vii).

The :SE2Comp correspondence node matches (viii), but its link to the position controller system element does not, because there is already a binding for the node se:SystemElement that binds a different object. So this must be repaired if this partial matching should be applied, denoted with the ⊛. Furthermore, the attribute constraint that ensures the equality of the system element's and the component's name must be enforced by changing the name of the pc:Component (again marked with a ⊛ at the first vertex (i) of the left subtree).

The right subtree represents the other partial matching from the previous example, where the ct:Controller, cd:CodeDescriptor, and ct:CodeContainer nodes could not be matched. Note that there are no real vertices for these unmatchable nodes in the tree. They are depicted dashed in Fig. 6 only to illustrate the repair operations needed to be performed to create a valid rule matching.

In fact, there is a third subtree (not shown in the figure). The search starts at every node of the root's matching (remember it contains bindings for all context and produced source graph nodes). Thus, starting from the se:SystemElement node, the algorithm would create this third subtree which contains the same matching as the second subtree, just in opposite direction.

Every vertex of the matching tree represents a possible repair operation. The number of reused elements is equal to the depth of a vertex in the tree (not the value of the *depth* counter), not counting the "step back to" vertices. Thus, using the root would not reuse deleted elements, but create the whole produced pattern. Once the tree is computed, it has to be decided which of the several partial matchings (i.e., which vertex of the tree) should be used. We have discussed above that a reasonable heuristic is to select the partial matching which does not damage reusable correspondences and which reuses most elements, i.e., will require the least repair operations. In this way, it is likely that only previously related elements are reused, which is probably the intention of the user.

Further details including the algorithm's pseudocode can be found in [7].

4.5 Runtime Evaluation

With our improved synchronization, we intend to address the issue of information loss during update operations, and focus less on performance improvements. Some operations of our solution turn out to be relatively time-consuming. Especially, building a complete search tree is exponential in the number of nodes and candidate objects (objects marked for deletion), but all techniques that calculate different matchings resp. repair alternatives will suffer from the general complexity of this problem. To estimate the performance impact, we implemented both our algorithm and the one by Giese and Wagner [5] in our TGG Interpreter[6]. Due to lack of space, we only give a short summary of the runtime evaluation here. Detailed results can be found in [7].

In summary, our algorithm works best when there are only few altered elements, because then the number of candidate objects is small. There could even be performance improvements when a large amount of revocations of dependent rules is prevented. Overall, the prevention of information loss comes with a performance decrease in most cases. However, in typical editing cases, the maximum performance drop was only 30 % in comparison with the old algorithm.

In our examples, we observed that good partial matchings were often found early in the partially reusable pattern search. Thus, additional heuristics could be used to determine the quality of a partial matching already during the search. When a good-quality matching is found, we could even decide to terminate the search, possibly long before the complete matching tree is build up. Then we may miss the intended way of reusing the elements, but we believe that there are many examples where adequate heuristics could determine the "best" matching early, improving the overall performance significantly. However, elaborating these heuristics is planned for future work.

[6] `http://www.cs.uni-paderborn.de/index.php?id=12842&L=1`

5 Related Work

Model synchronization has become an important research topic during the last years. Several concepts of incremental updates, which are mandatory for preserving model-specific information, have been proposed. However, as discussed in this paper, simply updating incrementally can still be insufficient. To the best of our knowledge, there are only few solutions that address these further issues.

As described in Sec. 4, the approach of Giese and Hildebrandt [4] is similar, but their main focus is performance and not optimizing the reuse of elements. Their approach is only able to cover cases similar to an element move (which means essentially repairing edges). It does not allow for more complex scenarios: either a rule application can be repaired by changing links or attributes value, or the rule is revoked and all elements are unrecoverably deleted.

Körtgen [10] developed a synchronization tool for the case of a simultaneous evolution of both models. She defines several kinds of damage types that may occur and gives abstract repair rules for these cases. At runtime, these general repair rules are used to derive concrete repair operations for a specific case. The synchronization itself, however, is a highly user-guided process, even if changes are propagated in just one direction. Our aim is to avoid unnecessary user interaction where that is possible. For ambiguous cases, however, we would also like to incorporate means for user interaction.

Xiong et al. [18] present a synchronization technique that also allows for the simultaneous evolution of both models in parallel. Basically, they run a backward transformation into a new source model, and then use model merging techniques to create the updated final source model. The same is done in forward direction. Other approaches (e.g., Jimenez [9]) also rely on model merging. In general, using model merging techniques in combination with (possibly non-incremental) transformations is another possibility to solve the issues discussed in this paper: a simple transformation propagates changes from the source model to a working copy of the target model. Then the model merger is responsible for merging the changes in the target model. However, this puts additional requirements on the model merger: first, it must identify "identical" elements without using unique IDs, because these IDs change when elements are recreated in a transformation. Second, discipline-specific information is lost in the working copy of the target model during a simple transformation, but still contained in the original target model. The model merger must be aware of this discipline-specific information in order not to overwrite it unintentionally. As model merging techniques evolve, it will be interesting to compare such techniques with our solution.

Another approach to the problem is using information on the editing operations that took place on a model. Ráth et al. [13] propose a solution which does not define the transformation between models any more, but maps between model manipulation operations. The problem with this approach is that a model transformation must be described in terms of corresponding editing operations, which may be a tedious and error-prone task.

Varró et al. [15] describe a graph pattern matching algorithm similar to ours. They also use a tree structure to store partial matchings. When (in-place) graph

transformation rules are applied, the matching tree is updated to reflect the changed graph, allowing the pattern matching itself to be incremental.

QVT-Relations [12] has a "check-before-enforce" transformation semantics which says that a pattern in the target model must be reused when there is an exact match with the target rule pattern. Nothing is reused if only parts of the target pattern can be matched. With this semantics, also QVT would delete and re-create the Mechatronic UML component in the above example. Even if we would change the semantics of QVT-Relations to allow for a better reuse of elements, algorithms for this semantics would yet have to be developed. Such an algorithm could use an approach similar to the one presented in this paper.

As mentioned before, performance was not in focus when developing the new synchronization. Therefore, it is likely that the heuristics and the search can still be improved. There exist several approaches for improving the performance of pattern matching. Varró et al. [16] use model-sensitive search plans that are selected during runtime. Especially when there are many elements marked for deletion that can possibly be reused, a dynamically selected search plan could also help increasing the performance in our case.

6 Conclusion and Future Work

We presented an improved incremental update mechanism that aims at minimizing the amount of unnecessarily deleted elements in the target model. In this way, much of the discipline-specific information that is not covered by the transformation can be preserved. The method is applicable not only for comparatively simple cases, like move operations, but also for more complex cases which involve alternative rule applications. One advantage of the technique is that it does not require the repair operations to be specified manually, but is a general solution and independent of the meta-models and the TGG rules.

The technique cannot prevent every possible inconsistency or loss of information. There are several improvements that we plan to investigate in the future. One extension is to involve the user if it is not clear how existing element have to be reused. Furthermore, the heuristic presented in Sec. 4 can be improved. For instance, we have described that the best partial match is the one that reuses most correspondence nodes. But as different TGG rules may have different correspondence node types, it is reasonable to reuse correspondence information even if the types do not match (and thus the correspondence object as such cannot be reused). Last, the technique presented here only considers one single rule application when building the partial matching search tree. The algorithm could be extended to find a best partial match over several rule applications and in this way further increase the amount of elements that are reused. This basically requires backtracking over rule applications.

The presented technique works only for changes in a single model. We are currently working on extending our TGG approach to support scenarios where two concurrently modified models must be synchronized.

References

1. Burmester, S., Giese, H., Tichy, M.: Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML. In: Assmann, U., Rensink, A., Aksit, M. (eds.) Model Driven Architecture: Foundations and Applications. Lecture Notes in Computer Science (LNCS), vol. 3599. Springer Verlag (2005)
2. Gausemeier, J., Frank, U., Donoth, J., Kahl, S.: Specification technique for the description of self-optimizing mechatronic systems. Research in Engineering Design 20(4), 201–223 (2009)
3. Gausemeier, J., Schäfer, W., Greenyer, J., Kahl, S., Pook, S., Rieke, J.: Management of Cross-Domain Model Consistency During the Development of Advanced Mechatronic Systems. In: Proc. of the 17th Int. Conference on Engineering Design (ICED'09) (2009)
4. Giese, H., Hildebrandt, S.: Efficient Model Synchronization of Large-Scale Models. Tech. Rep. 28, Hasso Plattner Institute at the University of Potsdam (2009)
5. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and Systems Modeling 8(1) (2009)
6. Greenyer, J., Kindler, E.: Comparing relational model transformation technologies: implementing Query/View/Transformation with Triple Graph Grammars. Software and Systems Modeling (SoSyM) 9(1) (2010)
7. Greenyer, J., Rieke, J.: An improved algorithm for preventing information loss in incremental model synchronization. Tech. Rep. tr-ri-11-324, Software Engineering Group, Department of Computer Science, University of Paderborn (2011)
8. Hearnden, D., Lawley, M., Raymond, K.: Incremental Model Transformation for the Evolution of Model-Driven Systems. Model Driven Engineering Languages and Systems (2006)
9. Jimenez, A.M.: Change Propagation in the MDA: A Model Merging Approach. Master's thesis, University of Queensland (2005)
10. Körtgen, A.T.: Modellierung und Realisierung von Konsistenzsicherungswerkzeugen für simultane Dokumentenentwicklung. Ph.D. thesis, RWTH Aachen University (2009)
11. Object Management Group (OMG): Meta Object Facility (MOF) Core 2.0 Specification (2006), http://www.omg.org/spec/MOF/2.0/
12. Object Management Group (OMG): MOF Query/View/Transformation (QVT) 1.0 Specification (2008), http://www.omg.org/spec/QVT/1.0/
13. Ráth, I., Varró, G., Varró, D.: Change-driven model transformations. In: Proc. of Model Driven Engineering Languages and Systems. Springer (2009)
14. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: 20th Intl. Workshop on Graph-Theoretic Concepts in Computer Science (1994)
15. Varró, G., Varró, D., Schürr, A.: Incremental Graph Pattern Matching: Data Structures and Initial Experiments. Graph and Model Transformation (2006)
16. Varró, G., Friedl, K., Varró, D.: Adaptive graph pattern matching for model transformations using model-sensitive search plans. Electronic Notes in Theoretical Computer Science 152 (2006)
17. Verein Deutscher Ingenieure: Design Methodology for Mechatronic Systems (2004)
18. Xiong, Y., Song, H., Hu, Z., Takeichi, M.: Supporting Parallel Updates with Bidirectional Model Transformations. In: Proc. of the 2nd Int. Conference on Theory and Practice of Model Transformations (ICMT '09) (2009)