Kanten zwichen den Hexas für TGGs und "halbe" Hexas.
Schön eigentlich.

A Precedence-Driven Approach for Concurrent Model Synchronization Scenarios using Triple Graph Grammars

Anonymous Author(s)

Abstract

Concurrent model synchronization is the task of restoring consistency between two correlated models after they have been changed concurrently and independently. To determine whether such concurrent model changes conflict with each other and to resolve these conflicts taking domain- or user-specific preferences into account is highly challenging. In this paper, we present a framework for concurrent model synchronization algorithms based on Triple Graph Grammars (TGGs). TGGs specify the consistency of correlated models using grammar rules; these rules can be used to derive different consistency restoration operations. Using TGGs, we infer a causal dependency relation for model elements that enables us to detect conflicts non-invasively. Different kinds of conflicts are detected first and resolved by the subsequent conflict resolution process. Users configure the overall synchronization process by orchestrating the application of consistency restoration fragments according to several conflict resolution strategies to achieve individual synchronization goals. As proof of concept, we have implemented this framework in the model transformation tool eMoflon. Our initial evaluation shows that the runtime of our presented approach scales with the size of model changes and conflicts, rather than model size.

CCS Concepts: • **Software and its engineering → Model-driven software engineering**; **Consistency**; *Software evolution.*

Keywords: bidirectional transformation (bx), concurrent model synchronization, triple graph grammars

ACM Reference Format:
Anonymous Author(s). 2020. A Precedence-Driven Approach for Concurrent Model Synchronization Scenarios using Triple Graph Grammars. In *Proceedings of ACM SIGPLAN International Conference on Software Language Engineering (SLE '20).* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

SLE '20, November 15–20, 2020, Chicago, IL

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Model-driven engineering [4] has proven to be an effective means to tackle the challenges that accompany the development of modern software systems, which are getting increasingly complex and distributed in nature. Often more than one model is needed to describe the developed software system from different, but overlapping perspectives. Keeping these models and various types of traceability relationships between them in a consistent state is a challenging task, often called model synchronization.

Model synchronization becomes especially challenging when multiple correlated models are changed concurrently. In such cases, not all changes can always be propagated between models as some may contradict each other and thus, are in conflict. This is the case, for example, when a change in one model leads to the deletion of elements in the other whose existence is the prerequisite for changes performed in that second model by another user. Yet, even for model changes that are not in conflict, there may be multiple ways to propagate them between models.

For a modern concurrent synchronization approach, it is of paramount importance to identify synchronization conflicts reliably and give modelers the ability to orchestrate model synchronization processes for guiding the process in accordance with their goals. Figure 1 gives an overview of a concurrent model synchronization process. Consistent interrelated models M_1 and M_2 are given and changed concurrently. The synchronization process identifies all conflicts between these changes and runs a conflict resolution process. The expected synchronization result is a consistent pair of models M_1'' and M_2'' that contains all conflict-free changes.

Some approaches provide solely one hard-wired solution of a conflict-detection and resolution strategy (from a universe of many different options), others come without any formal guarantees for their synchronization results or have an exponential runtime behavior w.r.t. the size of the processed models (cf. Section 2).

The contribution of this paper is a framework that simplifies the implementation (orchestration of a family) of concurrent synchronization algorithms with the following properties:

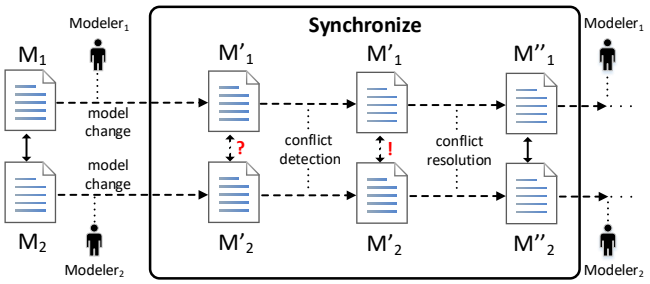

Figure 1. Synchronization Process

- These algorithms are derived from a declarative rule-based formal specification of a model consistency relation in the form of so-called Triple Graph Grammars (TGGs).
- They come with a number of predefined but extensible conflict-detection and -resolution as well as consistency restoration strategies.
- Formal properties can be shown using state-of-the-art category- and graph-transformation-based proof techniques (e.g., [11, 14, 21, 30])
- The intended scope of the effects and potential conflicts of model changes are identified using a TGG-based causal dependency relation for model elements.
- Scaleability with the size of processed model changes is guaranteed by limiting the effects of model updates to causally dependent areas in the regarded models and relying on incremental graph pattern matching techniques.

In Section 2, we give an overview of the state-of-the-art w.r.t. concurrent model synchronization approaches. Section 3 recalls various concepts related to TGGs. Sections 4 and 5 present our concurrent synchronization framework. The former explains in detail how conflicts are detected, while the latter presents strategies to resolve them and restore consistency. In Section 6, we introduce our implementation. Based on this, we evaluate our approach w.r.t. scalability in Section 7. Section 8 sums up our contribution and discusses future work.

2 Related Work

In this section, we discuss the current state-of-the-art in the field of concurrent model synchronization. Although we do not claim completeness of this survey, we are not aware of further works that differ fundamentally from the ones discussed here. The majority of the approaches in this field can be categorized into *propagation-based*, *constraint-based*, and *search-based* approaches. The approaches considered support *state-based* and *delta-based* model changes; however, they cannot be clearly clustered as some approaches abstract from the way how model changes are described and, therefore, support both state- and delta-based definitions of model changes.

State-based approaches [37–39] hold copies of all models in order to calculate differences, which is not only memory consuming, but also scales with the size of the involved models. In contrast, *delta-based* approaches [16, 17, 19, 35] operate on model changes, which may, e.g., be detected by an incremental pattern matcher. In general, delta-based approaches tend to scale better in scenarios with frequent changes, but require more bookkeeping, which may have negative effects to the memory consumption.

Propagation-based approaches to concurrent model synchronization use sequential synchronization steps to propagate the changes from one model to the other one followed by a propagation step in the opposite direction. All propagation-based approaches have severe drawbacks: Buchmann et al. [6] employ purely hand-crafted solutions, which do not guarantee any correctness. Especially, they do not show that the synchronization result is still in the given modeling language. Some approaches such as [6, 9, 25, 31, 34, 37–39] do not consider conflicts between changes on both sides and/or do not provide the means to identify and solve them, which can lead to problems when model changes are overwritten by propagation without asking the modeler. Some propagation-based approaches are able to tackle the problem of detecting conflicts between parallel updates such as [16, 17, 19, 22, 25, 35, 39] by analyzing if a propagation step contradicts with a model change. However, as shown by Orejas et al. [29], propagation-based conflict detection is not deterministic in general and may depend on the order in which changes are propagated. This means that, depending on the propagation order, certain conflicts may or may not be detected. Other approaches are limited to a confluent set of grammar rules [19] or are limited to specific kinds of models such as tree-like hierarchies [31].

Constraint-based approaches are often based on a relational specification which can be enforced using tools such as a SAT solver. They typically solve problems globally; this means that all possible synchronization solutions are encoded in a search space. The approach proposed by Macedo et al. [28], for example, finds the closest model that is consistent again. Closeness can either be defined w.r.t. graph edit distance or be based on user defined distance metrics to support user-preferences. However, this flexibility comes at the price of scalability as constraint-based approaches can often cope with rather small models only.

Search-based approaches explicitly explore and find a rich set of synchronization solutions, which can become very expensive with increasing search space. Cicchetti et al. [8] align their work with that by Macedo et al. in that they calculate all closest sub-models that are still consistent to a given relational consistency specification. Focussing on sub-models, there is a potentially large amount of information loss as their approach does not truly incorporate all kinds

of model changes. Orejas et al. [30] propose a TGG-based approach, where a set of consistency-describing grammar rules is used to find all possible parse trees of the given inter-related models and enriching them with annotations for, e.g., mandatory, removed, added, and no longer covered elements. These annotations are used to find conflicts, which are resolved using back-tracking to calculate all possible synchronization solutions and to present them to the modeler to choose from. However, finding all possible synchronization solutions is very expensive and the amount of presented alternative solutions to the user might be overwhelming.

In summary, all the approaches discussed have one or more limitations. We are looking for a concurrent synchronization approach that (1) does not come with severe restrictions concerning the structure of the processed models or the definitions of the regarded consistency relation, (2) finds all kinds of conflicts between concurrent model changes in a deterministic way, (3) allows the modelers to interact with the synchronization process, (4) reliably returns a synchronization result that belongs to the given modeling language, and (5) scales along the size of model changes and conflicts rather than along the model size. In this paper, we will present an approach that has all these properties. However, its scalability comes with the price of the restriction that consistency restoring operations modify only model parts that are causally dependent on those parts that are directly changed by modelers.

3 Triple Graph Grammars

In this section, we recall *triple graph grammars* (TGGs) [32], a declarative and rule-based approach to specify the consistency between two modeling languages. Being based on (typed attributed) graphs and their transformations as underlying formalism, TGGs are expressive and allow for the development of synchronization solutions with strong formal guarantees [11, 14, 21, 30]. Moreover, (many of) the operations needed during model synchronization algorithms can automatically be derived from the rules of a given TGG. Still, they can be implemented in a scalable way [1, 2]. As (typed attributed) graphs provide a suitable basis to formalize models and their transformations [3], TGG-based synchronization approaches are directly applicable to models. We thus use the terms "graph" and "model" interchangeably. In the following, the TGG concepts are recalled informally; a formal introduction can be found in, e.g., [10, 11]. An informal introduction to graph transformation [18] (including a chapter on model translation and synchronization) appeared recently. We illustrate TGGs and the basic ingredients for our synchronization approach using a running example.

Triple graphs and rules. A *triple graph* consists of three graphs: a *source graph*, a *target graph*, and a *correspondence graph* in between that connects source and target graphs via two graph homomorphisms. The correspondence graph serves to establish traceability links between correlated elements from source and target graphs. In practical applications, the underlying graphs are usually typed and attributed. During synchronization processes, the occurring objects may become *partial triple graphs* [13, 26]: A user may have deleted an element that was referenced by a correspondence morphism. Partial triple graphs still consist of three graphs; the graph homomorphisms connecting the correspondence graph with the source and target graph, however, may be partial, i.e., containing dangling references..

As a running example for a TGG, we define the consistency between a Java abstract syntax tree (AST) model (source) and a documentation model (target) as depicted in Fig. 2. This figure shows a metamodel (represented as a triple type graph) that declares the general syntax of models. The Java AST model consists of (Sub-)Classes containing Methods with Parameters and Fields, while the documentation model consists of Doc(ument)s with hyper references (href) to other Docs. Furthermore, a Doc contains Entries referencing Glossary Entries, that again are contained in a Glossary. Note that some elements have a name attribute, while Docs additionally store their version number. The correspondence types are depicted as hexagons referencing types of both, the Java and the documentation model, pair-wisely.

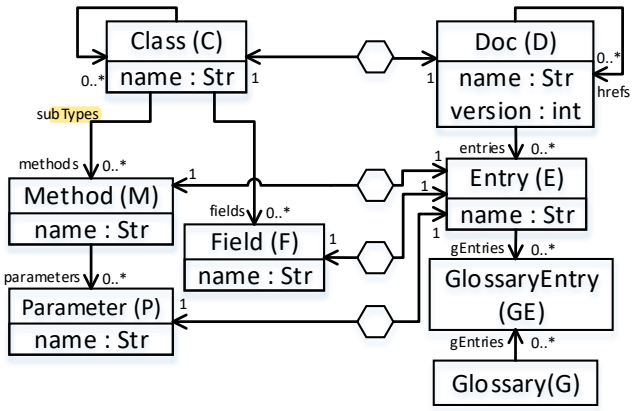

Figure 2. Metamodel of the running example

A *triple rule* consists of two triple graphs L and R (typed over the given triple type graph), called *left-hand side* (LHS) and *right-hand side* (RHS), respectively, such that their intersection $K = L \cap R$ is a triple graph again. Intuitively, the difference between L and K specifies all the elements to be deleted by an application of the rule, and the difference between R and K specifies all the elements to be created. Additionally, rules may be equipped with *negative application conditions* (NACs) that specify forbidden context in whose presence the rule is not applicable. Finally, we allow rules to be equipped with constraints concerning the attribute values. In this paper, we restrict them to be equations involving attribute values of corresponding elements only.

A *triple graph grammar* consists of a start graph (that is usually the empty graph) and a set of non-deleting rules, i.e., rules, where the LHS is a sub-triple graph of the RHS. The *language* defined by a TGG consists of all graphs derivable by an application sequence using its rules beginning at its start graph. Hence, a pair of models is consistent w.r.t. a given TGG iff a correspondence graph exists that extends the two models to a graph triple in the TGG language. Rewriting of partial triple graphs can be introduced analogously [26] and will be used to capture the semantics of synchronization operations manipulating dangling references.

Figure 3 depicts the rule set of our running example consisting of 8 TGG rules. They are displayed in an integrated fashion, i.e., as a single graph. The black, unmarked elements constitute the LHS of the rule, i.e., the context that has to exist for a rule to be applicable. Green elements annotated

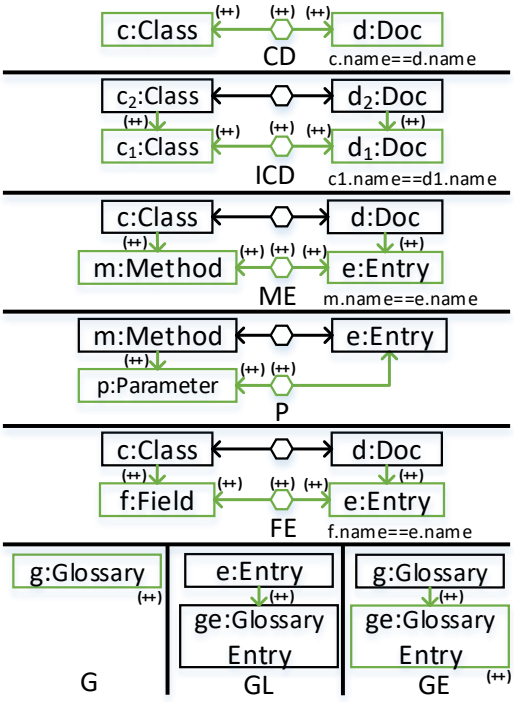

Figure 3. TGG Rules

with (++) are to be created when the rule is applied. The rule *CD* has no precondition and thus may be used to generate Classes with corresponding Docs arbitrarily often. The rule ICD creates a Sub-Class and a corresponding Doc with a hyper-reference when a Class with a corresponding Doc already exist. Note that we create the subClass link together with the sub-class, which implicitly forbids multiple inheritance. The rules *ME* and *FE* create Methods, resp. Fields, with corresponding Entries. Rule *P* creates a Parameter that corresponds to an already existing Entry. Finally, the rules *G*, *GE*, and *GL* create a Glossary together with Glossary Entries and links from Entries to Glossary Entries. These rules only act on the documentation model (the target side); the created elements do not have corresponding Java elements. Several rules (*CD*, *ICD*, *ME*, and *FE*) are equipped with attribute conditions. In each case, the condition declares that the names of the newly created elements should be equal. Figure 4 depicts a simple model that can be created applying first rule *CD* followed by applications of rules *ICD*, *ME*, and *P*.

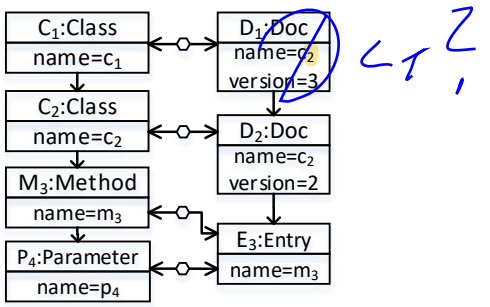

Figure 4. Exemplary Model

Rules derived from TGG rules. Triple rules can be used to create consistent models from scratch. For scenarios like *model translation* and *model synchronization*, suitable kinds of rules can be derived from the given TGG. First, a TGG rule can be *operationalized* to support forward (source → target) and backward (target → source) translations. Figure 5 depicts the forward operationalized rules CD_{FWD} and ICD_{FWD}, which are created by converting all green source elements to context first as we consider them to already exist. To prevent elements from being translated twice, we introduce annotations: □ → ☑ indicates that this element is still untranslated and applying the rule marks this element as translated. Consequently, ☑ indicates that the annotated element must be translated before applying this rule. Note that CD_{FWD} contains a NAC, depicted in blue and annotated with (nac), that forbids the Class *c* to be translated into a corresponding Doc when it is a Sub-Class of another Class. This kind of NAC is also called a *filter NAC* [20, 24] and adjusts the translation process to avoid dead-ends. If a Sub-Class is translated using CD_{FWD} (without that NAC), for example, there is not any other forward rule that translates the remaining link to its Super-Class. The creation of all other forward and backward rules can be done analogously.

Another useful operationalization of TGG rules is referred to as *consistency check rules*. They detect whether yet untranslated elements in a source and a target model can be considered as correlated. If so, they create correspondence links. While these kinds of rules have been used to compute (maximal) correspondence relations between previously unrelated models from the source and target domain [27], we employ them only *locally* to detect whether independently

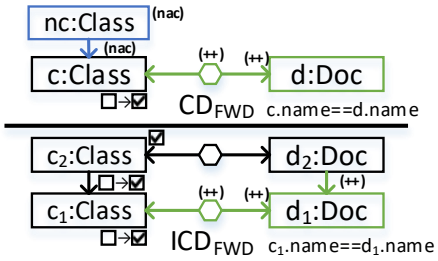

Figure 5. Exemplary TGG Forward Rules

added elements on source and target side can be considered as corresponding to each other. We refer to this process as *local-CC*. Figure 6 depicts an exemplary consistency check rule that is derived from rule *FE*. Moreover, we project these consistency check rules to their source and target parts only to obtain *source* and *target patterns* which we will use for conflict detection purposes later on.

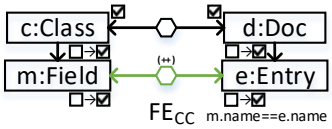

Figure 6. Exemplary TGG Consistency Check Rule

Finally, there are *short-cut rules* that are synthesized from TGG rules by means of a special kind of sequential rule composition operator [12]. Applying a short-cut rule replaces one TGG rule application by another one while allowing to preserve selected elements (instead of deleting and recreating them). Short-cut rules allow for advanced editing of models while preserving information in the process. Forward or backward operationalizing a short-cut rule results in *repair rules* that allow to directly propagate the edit the short-cut rule specifies on the source side to the target (and conversely). The crucial point is that short-cut rules specify complex (language-preserving) edits one cannot immediately perform using the original rules of the given TGG. Hence, their derived repair rules are often suitable to directly propagate "free" user edits. Short-cut rules do not need to be non-deleting and their derived repair rules may act on partial triple graphs. For details on their construction, conditions on language-preserving applications, and application in unidirectional model synchronization, we refer to [12, 13].

An example of a short-cut rule is given in Fig. 7, where the short-cut rule *CD-To-ICD* transforms an application of *CD* to one of *ICD*. *CD-To-ICD*$_{FWD}$ shows the forward operationalized short-cut rule, which directly propagates an edge that was newly inserted between two classes. Furthermore, also short-cut rules can be operationalized to obtain consistency check operations such as *CD-To-ICD*$_{CC}$.

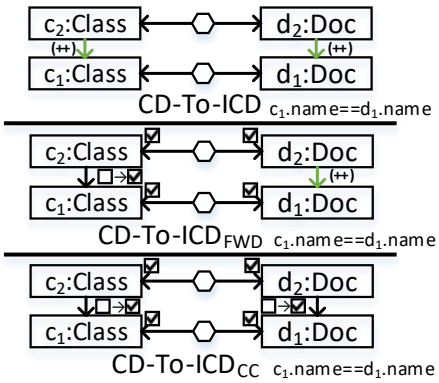

Figure 7. (Operationalized) Short-Cut Rule

Precedence graph. Given a consistent triple as depicted in Fig. 4, we can infer a *precedence graph* (PG) that describes with which TGG rule applications this triple can be derived and how those depend on one another. This means that a PG describes the aforementioned causal dependency relationship for model elements. Formally, a precedence graph is based on so-called *consistency patterns*. A consistency pattern is just the RHS of one of the rules of the given TGG. Applying a rule to a triple graph, one obtains a homomorphism from the RHS of the rule to the resulting triple graph: This homomorphism maps the elements of the LHS of the rule to the elements they have been matched to for applying the rule. The elements of its RHS that do not belong to the LHS (i.e., the elements to be created) are mapped to the newly created elements. Thus, given a sequence of rule applications, one obtains a family of homomorphisms from the rules' RHSs to the triple graph. These homomorphisms are in a natural *dependency relation*: One homomorphism is dependent on another one, if the former matches an element, the underlying rule application of the latter one creates. Moreover, these homomorphisms *cover* the triple graph in the following sense: Every element of it is matched exactly once by a rule element to be created, i.e., by an element of RHS \ LHS of one of the rules. We will also say that this match *explains* or *accounts for* the element. Conversely, given a TGG and some triple graph, if there is a family of homomorphism from the consistency patterns to this triple graph such that the dependency relation is acyclic and the family covers the triple graph in the above sense, the triple graph belongs to the language of the TGG (for a formalization and proof, we refer to [27, Lemma 4]). This is what we define to be a *precedence graph* (PG) for a triple graph M: an acyclic graph where the nodes are homomorphisms from consistency patterns to M such that M is covered by them, and edges are their dependencies. Note that the dependency relation stored in a PG induces a causal dependency relation on elements of the triple graph: An element x (node or edge) depends on an element y if the element y is matched by the rule that

creates x. Similar information (about dependency between and coverage of elements) has been used by Kehrer to lift atomic model changes to the level of *edit scripts* [23].

Figure 8 depicts the PG for the model in Fig. 4 with each node corresponding to a TGG rule application where the name is based on the TGG rule name and an index representing the indices of created elements. The boxes inside the nodes represent the state of created elements of the corresponding rule application on source (left box) and on the target (right box) side. In the next chapter, we will introduce annotations for these boxes that will help us detecting conflicts.

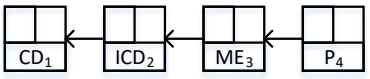

Figure 8. Exemplary Precedence Graph

4 Conflict detection

In this section, we present our approach to the detection of conflicts during model synchronization. We assume the following general setting (compare Fig. 1 again): A TGG is fixed; it defines a consistency relation between two modeling languages. Our concurrent synchronization process starts with a pair of consistent models M_1, M_2, or, somewhat more formally, a triple graph belonging to the language of the given TGG (i.e., M_1 is the source and M_2 the target graph). This pair of consistent models comes with a precedence graph *PG*.

Both models are changed independently by two modelers, resulting in models M_1' (source graph) and M_2' (target side). Compared to M_1, in M_1' some elements may have been added, some deleted, and some attribute values may have changed; the same holds for M_2 and M_2'. We call this change a *(model) delta*, and speak of *source* or *target delta* when we want to refer to only one of these changes. We do not make any assumptions on how they have been performed; we only assume that there is a way to identify the remaining elements of M_1 and M_2 with their counterparts in M_1' and M_2', respectively.

Our goal is to find models M_1'' and M_2'' that are consistent, i.e., which are source and target graphs of a triple graph of the given language. Moreover, M_1'' and M_2'' should not differ too much from M_1' and M_2', respectively. In general, there is no unique solution for this problem, even if requiring the distance between (M_1'', M_2'') and (M_1', M_2') to be *minimal* (according to some metric). Therefore, we provide modelers with the possibility to orchestrate the synchronization process, leading to individually defined outcomes.

To compute a pair of consistent models, we first extend and annotate the precedence graph to obtain a *delta precedence graph* (DPG). Basically, this graph comprises information which parts of the PG have been affected by the deltas and how the individual changes can (locally) be propagated to the respective other side. *Our guiding principles for synchronization are to only change the models where the delta makes this necessary and to preserve as many of the model changes as possible, e.g., to not delete newly added elements and to not recreate deleted ones* (compare, e.g., [7, 33] for these principles). Whenever it is not possible to simultaneously preserve user-changes on the source and on the target model such that the model remains in the language of the TGG, we call this a *conflict*. Our synchronization process first analyzes the delta precedence graph for conflicts and subsequently, propagates the source and target deltas to the respective other side while resolving the detected conflicts according to an orchestration given by the user.

In this section, we introduce delta precedence graphs and different types of conflicts illustrated using our running example.

Figure 9 shows an example of a concurrent model change applied to a formerly consistent model graph. In the source graph of the original model, there are two Classes (C_1 and C_2) that have a Field and Method each. Furthermore, M_6 has two Parameters and M_8 has one. The target graph has two Docs (D_1 and D_2) that contain two Entries each, where the Entries contained in D_2 reference the Glossary Entry GE_{13}. GE_{13} and GE_{12} are contained in the Glossary G_{14}. Elements with the same indices on both sides have a correspondence link, except for Parameters, which are connected to those Entries that correspond to their Method.

Several concurrent *model changes* have taken place with different impacts and issues. First, the Field F_7 is pulled up into a newly created Super-class C_3 of C_2. In the target graph, the corresponding Entry E_7 is moved to D_1, which does not correspond to C_3. Also, a new Field F_4 and a new Entry E_4 are created within elements C_1 and D_1, respectively. They have the same names. Additionally, M_6 is deleted together with its Parameter P_9, while the Parameter P_{10} was moved to M_8. However, while M_6 is deleted in the source graph, the corresponding target element E_6 as well as E_4, E_5, and E_7 are linked to GE_{12}. Finally, the names of corresponding model elements M_8 and E_8 are changed to different values. Similarly to our notation of rules, in Fig. 9, newly added elements are depicted in green and marked with (++), deleted ones in red and marked with (−−), and attribute changes are indicated via an arrow →.

4.1 Delta Precedence Graphs

A *delta precedence graph* extends a precedence graph by information on how a delta affected the validity of the precedence graph. For elements added in the delta, we need to find suitable TGG-rule applications that could have created these elements; potentially correlating elements have to be created on the other side. Moreover, consistency matches of

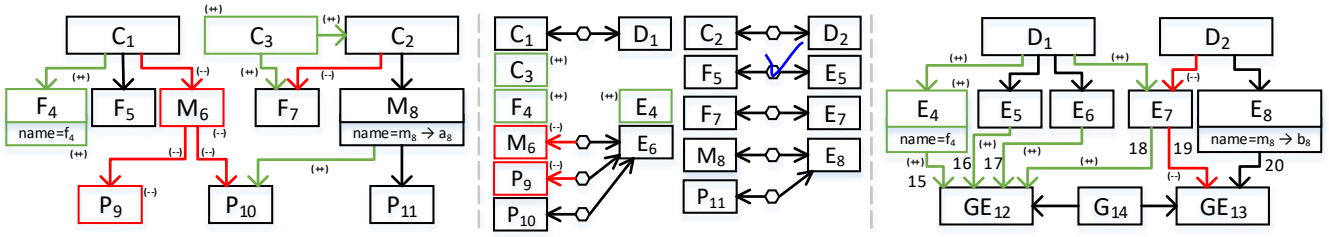

Figure 9. Running Example – Model and Delta

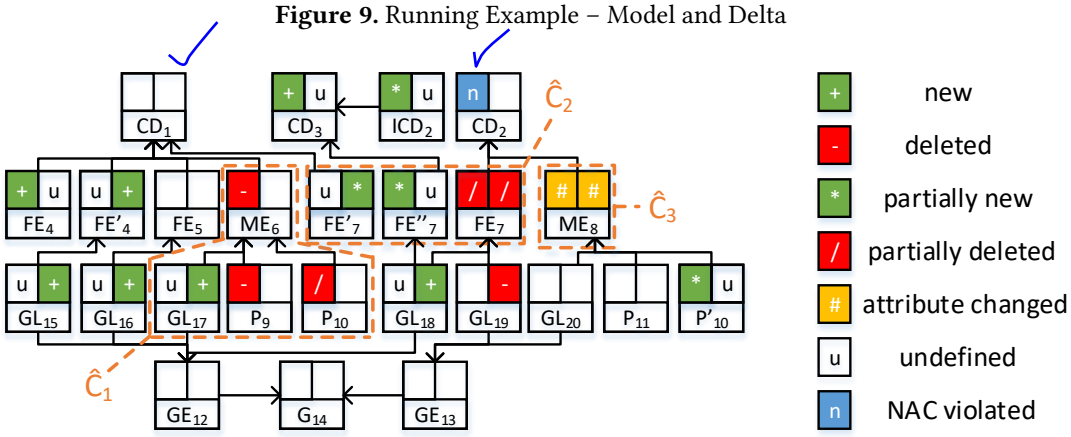

Figure 10. Running Example – Delta Precedence Graph

the precedence graph may have been invalidated (or "broken") in three different ways: (i) attribute values have been changed such that an attribute condition of a TGG rule is violated, (ii) elements that are covered by a consistency match have been deleted, and (iii) elements have been added so that already existing elements have to be parsed anew since a NAC is violated. Elements that are no longer covered by a consistency pattern since the formerly covering one is "broken" have to be matched anew as well. All this information is collected in a DPG. In the definition of DPG, we call an element of the updated model *unpropagated* if one of the following two cases applies:

1. This element has been newly added by one of the user edits, or
2. there is a consistency pattern that explains how this element has been created (i.e., the underlying rule application created this element). However, this rule application has been rendered invalid by the user edits (either, because one of its filter NACs is violated now or because a matched element has been deleted).

Definition 4.1 (Delta precedence graph)**.** Let a TGG and a pair of consistent models (M_1, M_2) together with a precedence graph PG for it be given. The *delta precedence graph* (DPG) for (M_1, M_2), PG, and a delta (consisting of sequences of graph changes leading from M_1 to M_1' and M_2 to M_2', respectively) consists of the nodes of the given precedence graph PG and a set of new nodes consisting of matches for source and target patterns that are such that at least one unpropagated element is matched by an element the underlying rule of the pattern creates.

Edges are, again, defined via dependencies.

Moreover, the nodes of the DPG have *source* and *target annotations* over the alphabet $\{+, -, *, /, \#, u, n\}$ according to the following rules:

- Nodes stemming from the original PG, i.e., nodes indicating formerly valid rule applications, are annotated on the source side in the following way:
  - annotation "–" whenever all elements the underlying rule application created on the source side have been deleted by the source edit;
  - annotation "/" whenever some (but not all) elements the underlying rule application created on the source side have been deleted by the source edit or if the match for this consistency pattern is broken because context elements are missing;
  - annotation "#" whenever some attribute values were changed by the source edit such that at least one attribute constraint of the underlying rule application is violated;
  - annotation "n" whenever the source edit added a new element that introduces a violation of a source NAC of the forward rule of the rule from which the consistency pattern is derived.

The target annotation is defined completely analogously. A node may also have more than one such annotation, which is why we will refer to sets of annotations in the following. However, for simplicity reasons, nodes in our running example only have sets of size 1.

- Whenever all nodes that the underlying rule created on the source side are matched to unpropagated elements, a node corresponding to a match for a source pattern is annotated with "+" on the source and with "u" on the target side. Symmetrically, the annotations for such a target pattern are switched.
- Whenever at least one node on the source side, created by the underlying rule, is matched to an already propagated element that was part of another formerly intact consistency match, a node corresponding to a match for a source pattern is annotated via "$*$" on the source and "u" on the target side. Symmetrically, the annotations for such a target pattern are switched.

Figure 10 depicts the DPG that corresponds to the model changes described above. One of the nodes FE_7, for example, is marked with "/" on both source and target side because the edges created by the corresponding rule application have been deleted on both sides; however, nodes F_7 and E_7 are preserved. The two new nodes indexed with FE'_7 and FE''_7 denoting a new source and target pattern match, respectively, indicate by which rule the now unaccounted nodes F_7 and E_7 could possibly have been created. The consistency restoration has to check whether they can be combined into a single TGG-rule application creating the two correlated elements simultaneously. Node ME_8 is annotated with "#" on both sides as the attribute values have been changed on both sides and the constraint requiring equal names is violated now. As a last example, the node CD_2 is annotated with "n" on the source side: Due to the newly introduced inheritance edge, it is no longer possible for the class C_2 to be created using rule *CD*. Thus, this node becomes unpropagated; the node indexed with ICD_2 indicates a new possibility to parse this node.

4.2 Conflicts

As discussed above, the annotations of a DPG indicate some synchonization actions that have to take place; as long as the dependency is respected (i.e., is still acyclic), a consistent triple graph is restored as soon as every annotation has been dealt with. Our definition and treatment of conflicts is based on the already mentioned idea of change-preservation. Whenever a user edit directly affected an element, this effect should be preserved. This means that newly created elements or elements whose attribute values have been changed are *intended to persist*. Deleted elements are intended to remain deleted. A conflict is a situation where all options available to propagate a certain change require to undo another one. We classify such conflicts based on the annotations occurring in DPGs and use the term *conflict scope* to refer not only to a conflict but also to other elements that depend on how the conflict is resolved. In the following, we assume a pair of models that was originally consistent, a PG for it, a model delta, and the induced DPG to be given. We illustrate all kinds of conflicts using our example in Fig. 10 (where the *conflict scopes* $\hat{C}_1$, $\hat{C}_2$, and $\hat{C}_3$ are indicated by dashed orange lines).

Preserve-delete conflict. A preserve-delete conflict is a situation where one of the deltas deletes a certain element whereas its corresponding element is used in the other delta and thus, intended to persist: A *potential preserve-delete conflict* is a node of the DPG where "−" or "/" belongs to the source or target annotation (except for the case where both annotations are "−"). It is a *preserve-delete conflict* if there is a newly added element on the other side or an element whose attribute value has been changed such that, by propagating the deletion, all patterns that would have been able to create that element vanish. Its *scope* is the node itself and all nodes transitively depending on it.

$\hat{C}_1$ depicts a *preserve-delete conflict* that is characterized by a full deletion ("−") in one precedence node on the source side and a creation of new elements ("+") in one of its dependent nodes on the target side.

Here, M_6 is deleted on the source side, but a new reference is added between E_6 and GE_{12} on target side. The conflict scope $\hat{C}_1$ includes changes that must not directly be in conflict with each other, but have to be considered when resolving the conflict, e.g., non-revoking P_9 implies non-revoking M_6.

Correspondence preservation conflict. A correspondence preservation conflict is a situation, where deltas modify corresponding elements on both source and target side such that it is not possible to restore the consistency without either deleting the correspondence relationships between the affected elements (and creating new correspondence relationships to different elements, which in the general case have to be created, too) or without discarding changes in the source or target model. In our example, $\hat{C}_2$ is such a conflict where F_7 is moved to C_3 while E_7 is moved to D_1. However, C_3 and D_1 do not correspond to each other and thus there is no TGG consistency pattern that can correlate these changes. Hence, we have to decide whether to revoke the relocation of either F_7 or E_7.

Every node of the DPG where "n" or "/" belongs to the source and target annotation is a *potential correspondence preservation conflict*. It is a *correspondence preservation conflict* if there exists no intact TGG consistency pattern match that again covers and relates these elements. Its *scope* is the node itself as well as nodes with a "$*$" annotation whose

corresponding rule applications create some elements that are also created by the rule application of the node itself.

Attribute change conflict. An attribute change conflict is a special case of a correspondence preservation conflict, where attribute values of until now corresponding source and target elements have been changed in such a way that the attribute values on both sides are no longer consistent. In our example, $\hat{C}_3$ is an attribute change conflict that occurs in node ME_8. The name of M_8 is changed from m_8 to a_8, while that of E_8 is changed from m_8 to b_8.

If both names would have been changed equally, no conflict would have been detected and the incremental pattern matcher would not detect any broken rule application. Every node of the DPG where "#" belongs to the source and the target annotation is a *potential attribute change conflict*. It is an *attribute change conflict* if furthermore an attribute constraint is violated because both attribute values were changed.

In Appendix A, we provide a overview of how the annotations of a node in a DPG relate to possible conflicts.

5 Consistency restoration

In this section, we will present a catalog of concurrent synchronization fragments, which can be orchestrated and executed sequentially to restore the consistency of the model under change in a concurrent synchronization scenario. This catalog includes the previously introduced operationalizations of TGG rules and three pre-defined conflict resolution strategies. Users can orchestrate their specific consistency restoration processes using these fragments.

In the following, we present all the concurrent synchronization fragments of our catalog, show an example orchestration of them which is quite typical, and apply this orchestration to our running example. All the following fragments except for *Resolve Conflict* are applied to elements only that do not belong to a conflict. *Resolve Conflict* is applied to each conflict (scope). Furthermore, if a fragment is applied, it processes all feasible elements until no unprocessed one can be found.

- ***Local CC*** is used to find and correlate newly added elements in the source and target graphs that may correspond to each other. This fragment processes related pairs of precedence nodes annotated with **(+|u)** and **(u|+)**. If it is chosen, it has to be applied before *Translate* as this also processes newly added elements that are no longer available afterwards.
- ***Translate*** is used to translate newly added elements to the opposite side and thus complete the rule application by applying a forward or backward operationalized TGG rule. This fragment processes precedence nodes annotated with **(+|u)** and **(u|+)**.
- ***Repair*** employs short-cut rules to fix broken precedence nodes. Forward and backward operationalized short-cut rules propagate complex changes from one side to the other. Consistency check operationalized short-cut rules allow to find corresponding changes on both sides and resolve them if possible. This fragment processes precedence nodes that are annotated with **"n"** or **"/"** on one or both sides and related nodes annotated with **(*|u)** or **(u|*)**.
- ***Resolve conflict*** is applied to each conflict (scope) and can be configured for each type of conflict that we identified in the previous chapter. We can call *Translate*, *Repair* and *Propagate*, where *Repair* can be used to reduce the conflict size beforehand, while *Translate* and *Propagate* can only be used after the conflict (scope) has been resolved because both include propagation steps that can only be executed afterwards. For resolving a conflict (scope), we offer three pre-defined strategies:
  - ***Take Source*** discards all the changes on the target side.
  - ***Take Target*** discards all the changes on the source side.
  - ***Preserve*** discards all the deletions that block newly added elements from being propagated.

  All applied fragments within *Resolve Conflict* are applied to elements only that belong to the current conflict (scope).
- ***Rollback*** revokes rule applications in cases where all the deletions were performed consistently on both sides or only on one side and the other side remains untouched, e.g., all green source elements were deleted. These nodes are annotated solely with **"-"** on one or both sides.
- ***Propagate*** applies *Repair* first to fix broken matches rather than revoking them. Then, it applies *Rollback* to revoke rule applications that have been consistently deleted on one side. Finally, *Translate* is applied and translates newly added elements to the opposite side. From our experience, calling these fragments in that specific order is a good choice for another fragment as it yields a sequential synchronization control flow.
- ***Clean up*** deletes all elements from source, correspondence and target graphs that are currently inconsistent w.r.t. our TGG. This fragment can only be called at the end of a synchronization process but can also be omitted if the user decides to eliminate inconsistencies in this way later on.

To restore the consistency of the model in our running example, we choose the following orchestration of fragments to be applied in the given sequential order: *Local CC* → *Translate* → *Repair* → *Resolve Conflict* {Repair → Take Source → Propagate} → *Propagate* → *Clean up*. For simplicity reasons, we resolve all three conflicts ($\hat{C}_1$, $\hat{C}_2$, $\hat{C}_3$ from Fig. 10) uniformly.

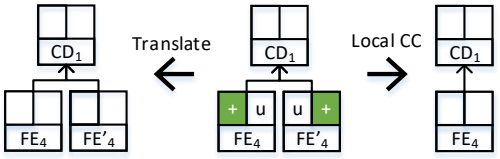

Figure 11. Running Example – Translate vs. Local CC

To correlate corresponding changes on both sides, we apply *Local CC* first. Nodes F_4 and E_4 are created independently with the same name but not set into correspondence yet. Using *Local CC*, we are also able to relate both changes to each other, creating the missing correspondence link in between. The effect on the precedence graph is shown on the right of Fig. 11. On the left of this figure, we see the result of not using *Local CC* but *Translate*. Applying *Translate* on both F_4 and E_4 independently of each other would create corresponding elements on the opposite sides using the proper forward and backward rule. This step results in new elements F_4' and E_4' besides the former ones.

Next, we apply *Translate* to several changes that are not contained in a conflict (scope) (as described in Section 4). This is the case for CD_3, GL_{14}, and GL_{15}. Translating newly added elements establishes consistency of each precedence node since the underlying rule applications are now complete and thus consistent.

Afterwards, we apply *Repair* to fix broken rule applications such as CD_2 that is inconsistent due to a NAC violation. Figure 7 depicts the shortcut rule CD-To-ICD$_{FWD}$ that can be applied here to transform CD_2 into ICD_2. The effect is that D_2 is preserved and an edge between D_3 and D_2 is created . The corresponding effect on the precedence graph is shown in Fig. 12. Note that the prior propagation of CD_3 is necessary to create the context needed by CD-To-ICD$_{FWD}$ to be applicable.

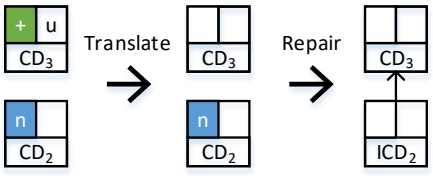

Figure 12. Running Example – Translation then Repair

Now, only conflicts (scopes) are still there, which are resolved in any order using *Resolve Conflict*. The intermediate model and delta precedence graph can be found in Appendix C. Starting with conflict scope $\hat{C}_1$, the primary issue is that a glossary link was created at entry E_6 while deleting the method M_6 that corresponds to E_6. Considering the precedence node P_{10}, only parts were deleted which implies that some of the remaining elements are to be preserved (here P_{10}). Using *Repair* first allows us to reduce the size of $\hat{C}_1$ by propagating the re-location of P_{10} from M_6 to M_8. This step is depicted in Fig. 13.

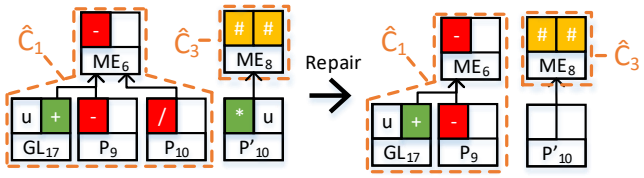

Figure 13. Running Example – Reduce conflict scope $\hat{C}_1$

Figures 14 (b) – (d) depict the results of all three conflict resolution strategies applied to conflict scope $\hat{C}_1$ in Figure 14 (a). The application of *Preserve* is of special interest as it revokes deletion deltas that block create deltas from being propagated. Following that strategy, the changes to ME_6 are revoked to solve the conflict, while keeping the changes to P_9 untouched. Applying any of these strategies leaves the remaining elements in a state where they can be propagated without colliding with any changes on the opposite side. To resolve $\hat{C}_1$ finally, we use *Take Source* (as specified earlier) and revoke all changes on the target side that are related to the conflict.

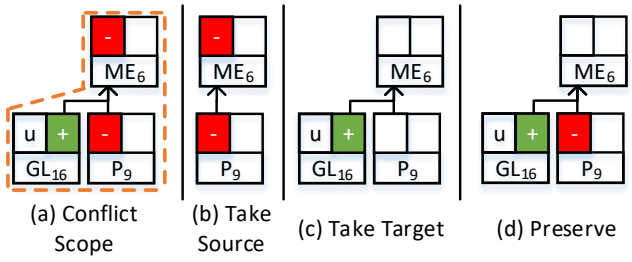

Figure 14. Running Example – Resolving $\hat{C}_1$

For resolving the conflict scope $\hat{C}_2$, we perform *Repair* first, but this application does not have any effect here as both changes contradict each other. Figure 15 depicts the results of *Take source* (b) and *Take target* (c), which revoke the changes on one side. Using the operationalized shortcut rule *CD-To-ICD*$_{FWD}$ after (b) or *CD-To-ICD*$_{BWD}$ after (c) propagates the changes to the opposite side, respectively. Note that *Preserve* has no effect here due to the nature of *correspondence preservation* conflicts. Since we chose *Take Source* as conflict resolution strategy, we can react to the relocation of F_7 to C_3 now by also moving E_7 from D_2 to D_3. This is done by applying *Propagate*.

Finally, we apply *Repair* to conflict scope $\hat{C}_3$, which has no effect as both attribute changes contradict each other. Resolving $\hat{C}_3$ can be done by choosing either *Take source* or *Take target*. Again, *Preserve* would not have any effect since there

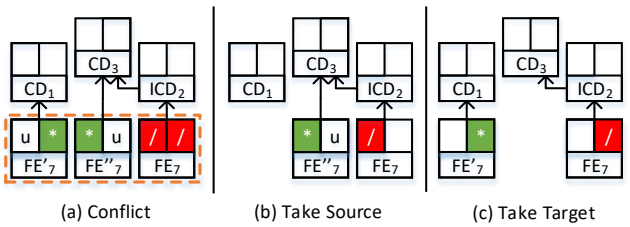

Figure 15. Running Example – $\hat{C}_2$

are no deletions that block the propagation of additions. Instead, we have to decide which attribute change to propagate (which implies to revoke the opposite one). Applying *Take Source* and subsequently *Propagate*, the remaining attribute change is propagated by re-evaluating the corresponding attribute constraint and transferring the value to E_8.

This leaves *Clean up* with nothing to do since all changes have been accounted for. The final model and delta precedence graph are depicted in Appendix C.

6 Implementation

Our approach is implemented in a synchronization component as part of the state-of-the-art model transformation tool eMoflon [36]. Figure 16 depicts the synchronization components with its interdependencies to an incremental pattern matcher and its inputs and outputs. In our synchronization framework, we allow modelers to change the source and target of a triple graph independently. eMoflon keeps track of these changes by employing an incremental pattern matcher that throws events when new matches of TGG rules have been detected or existing ones have been invalidated. This information is used to update a (delta) precedence graph which represents the dependencies between TGG rule applications. The synchronization component analyzes the delta precedence graph to detect conflicts. They can be resolved by a synchronization and conflict resolution orchestration that has been implemented by an integration manager, who is an expert in both source and target domain. However, we offer pre-defined configurations such as the one from our running example in the previous chapter that can be extended. Applying this orchestration resolves the previously detected conflicts step-by-step and restores consistency of the triple graph. eMoflon already has an extensive test suite[1] with 344 tests for various TGG-based consistency restoration scenarios from which 25 constitute concurrent synchronization tests. In the near future, we will extend this test suite especially w.r.t. to more concurrent synchronization scenarios.

7 Evaluation

In this section, we will present our evaluation results, which are based on our implementation in eMoflon and two different TGG projects. The first TGG is our running example

[1] https://github.com/eMoflon/emoflon-ibex-tests.

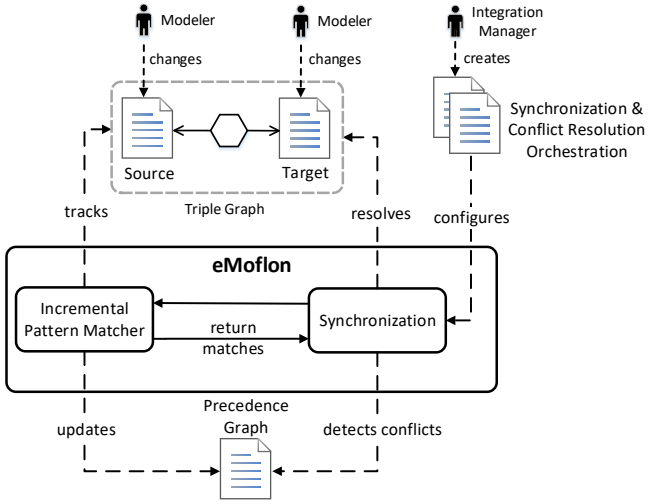

Figure 16. eMoflon - Synchronization Components

consisting of 8 TGG rules, while the second one is based on the example introduced in [13] and consists of 28 TGG rules that define consistency between MoDisco [5] and custom documentation models. However, the second TGG does not only contain more rules but also more asymmetric rules, i.e., rules that are no simple 1-to-1 mapping between source and target. As a test environment, we use a workstation with an AMD Ryzen Threadripper 2990WX 3GHz 32xCore using 128GB of RAM. One of the main goals of our approach is to provide a scalable concurrent synchronization solution. Therefore, we pose the following research questions: **RQ1:** *Does our synchronization approach scale with the size of the model or with the number of changes and conflicts?* **RQ2:** *Does the performance of the conflict detection change if less changes lead to actual conflicts?* **RQ3:** *How does the number of changes and conflicts affect the conflict resolution performance?*

To answer these questions, we investigate two scenarios for each TGG: First, we generate a fixed number of 100 conflicts and increase the size of both, the source and the target model, which we measure in number of nodes. Second, we choose a fixed model size of about 500 000 nodes (in the source and target models together) and increase the number of concurrent changes. To investigate the correlation between changes and actual conflicts, we distinguish between 4 sub-scenarios by choosing the changes in such a way that 25%, 50%, 75% and 100% lead to an actual conflict, respectively. For all scenarios, we plot the initialization time where the incremental pattern matcher collects all matches for the still consistent triple graph, i.e., computes the precedence graph. Then, we apply a number of changes to both, the source and target model and measure the time to restore consistency. Note that each change is meant to induce one of the three conflict types previously presented. For each data point, we measured 20 repetitions and took the average

value over all runs. Note that the plots of the second example can be found in Appendix B.

Figure 17 depicts the plot of the first scenario and our running example: The initialization time increases for this TGG linearly with the size of the model, while the conflict resolution time stays almost constant.[2] The same holds for the second TGG with a larger set of rules, which takes about 19 % longer to initialize. This means that the model size does not directly affect the performance of conflict resolution **(RQ1)**. Figure 18 depicts the plots for the second scenario and our running example: The initialization time stays constant with a constant model size of 500 000 (source and target) nodes while the time to detect and resolve conflicts increases linearly. Whether the changes to both sides are in conflict with each other has only a minor impact on the performance and increases the gradient slightly. For the larger set of rules, the impact becomes more significant and takes 16 % more seconds per 25 % more conflicts **(RQ2)**. Thus, we can conclude that the performance scales linearly with the size of changes and to some point with the amount of conflicts that are introduced by changes **(RQ3)**.

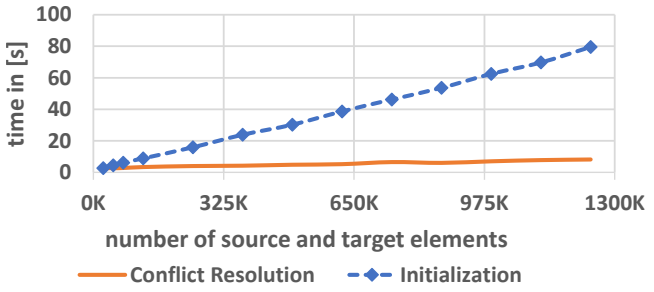

Figure 17. Increasing model size with constant number of 100 conflicts

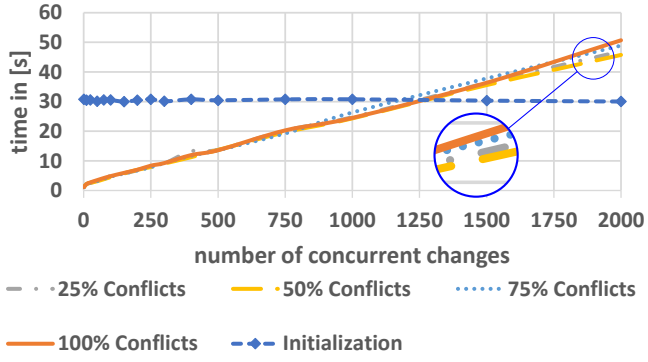

Figure 18. Increasing number of changes with constant model of 500K nodes

[2]It slightly increases since deletions become more expensive with increasing model size in the Eclipse Modeling Framework.

Threats to validity. Our evaluation is based on synthesized models and changes only. Thus, it remains future work to investigate real-world scenarios by analyzing, e.g., Git merge requests and deducing models from code. Furthermore, we evaluated only with two TGGs that have different characteristics but describe a similar scenario. However, the rules of both TGGs are symmetric as well as asymmetric and thus, do not only represent simple 1-to-1 mappings, which makes them representative for a broader range of TGGs.

8 Conclusion

In this paper, we presented a scalable TGG-based, precedence-driven concurrent model synchronization approach. We showed how to use a delta precedence graph to identify conflicts and introduced a modular framework that enables developers to resolve these conflicts and, furthermore, orchestrate the whole process to achieve specific synchronization goals. Giving a broad overview of the landscape of concurrent synchronization works, we showed that this approach is indeed novel in that we detect conflicts before propagating any changes. Furthermore, we showed how to detect a new kind of conflict, namely *correspondence preservation* conflicts, which to the best of our knowledge, no other approach is able to detect so far. Our approach has been implemented and evaluated within the state-of-the-art graph transformation tool eMoflon. In the evaluation, we showed for two different TGG projects that our synchronization approach scales linearly with the size of changes instead of the model size.

For the future, we plan to extend and formalize our approach w.r.t. handling further types of conflicts between rule applications that, if applied, would violate, e.g., multiplicity constraints of a metamodel. Further investigations are also needed to study the effects of one major design decision of our concurrent model synchronization approach in practice: still consistent rule applications remain untouched if they do not (causally) depend on a broken rule application. This design decision is inspired by *a least change and least surprise principle* [7] and is one main reason for the scaleability of our approach. Due to this, our synchronization algorithm is in general not always able to reestablish the consistency of two models when a change in one model requires its propagation against the introduced causal dependency relation in the related other model. However, there are works such as [15, 24] that show how to derive application conditions with a statically analyzable criterion for TGGs such that our synchronization algorithm very rarely runs into such a situation. In our experience TGGs violate this criterion if and only if the propagation of a local change in one model would have rather unexpected and unwanted global effects on the other model from a user's point of view. But further experiments are needed to confirm our experiences. Finally, we plan to build up a rich zoo of concurrent synchronization scenarios and to compare different approaches with respect to these.

References

[1] Anthony Anjorin, Thomas Buchmann, Bernhard Westfechtel, Zinovy Diskin, Hsiang-Shang Ko, Romina Eramo, Georg Hinkel, Leila Samimi-Dehkordi, and Albert Zündorf. 2020. Benchmarking bidirectional transformations: theory, implementation, application, and assessment. *Journal of Software and Systems Modeling* 19, 3 (2020), 647–691. https://doi.org/10.1007/s10270-019-00752-x

[2] Anthony Anjorin, Zinovy Diskin, Frédéric Jouault, Hsiang-Shang Ko, Erhan Leblebici, and Bernhard Westfechtel. 2017. BenchmarX Reloaded: A Practical Benchmark Framework for Bidirectional Transformations. In *Proceedings of the 6th International Workshop on Bidirectional Transformations co-located with The European Joint Conferences on Theory and Practice of Software, BX@ETAPS 2017, Uppsala, Sweden, April 29, 2017 (CEUR Workshop Proceedings)*, Romina Eramo and Michael Johnson (Eds.), Vol. 1827. CEUR-WS.org, 15–30.

[3] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. 2012. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Journal of Software and Systems Modeling* 11, 2 (2012), 227–250. https://doi.org/10.1007/s10270-011-0199-7

[4] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-Driven Software Engineering in Practice, Second Edition*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00751ED2V01Y201701SWE004

[5] Hugo Brunelière, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. 2014. MoDisco: A model driven reverse engineering framework. *Journal of Information and Software Technology* 56, 8 (2014), 1012–1032. https://doi.org/10.1016/j.infsof.2014.04.007

[6] Thomas Buchmann and Sandra Greiner. 2016. Handcrafting a Triple Graph Transformation System to Realize Round-trip Engineering Between UML Class Models and Java Source Code. In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 2: ICSOFT-PT, Lisbon, Portugal, July 24 - 26, 2016*, Leszek A. Maciaszek, Jorge S. Cardoso, André Ludwig, Marten van Sinderen, and Enrique Cabello (Eds.). SciTePress, 27–38. https://doi.org/10.5220/0005957100270038

[7] James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. 2017. On principles of Least Change and Least Surprise for bidirectional transformations. *Journal of Object Technology* 16, 1 (2017), 3:1–31. https://doi.org/10.5381/jot.2017.16.1.a3

[8] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. 2010. JTL: A Bidirectional and Change Propagating Transformation Language. In *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers (Lecture Notes in Computer Science)*, Brian A. Malloy, Steffen Staab, and Mark van den Brand (Eds.), Vol. 6563. Springer, 183–202. https://doi.org/10.1007/978-3-642-19440-5_11

[9] Alexander Egyed. 2007. Fixing Inconsistencies in UML Design Models. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 292–301. https://doi.org/10.1109/ICSE.2007.38

[10] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. 2006. *Fundamentals of Algebraic Graph Transformation*. Springer. https://doi.org/10.1007/3-540-31188-2

[11] Hartmut Ehrig, Claudia Ermel, Ulrike Golas, and Frank Hermann. 2015. *Graph and Model Transformation - General Framework and Applications*. Springer. https://doi.org/10.1007/978-3-662-47980-3

[12] Lars Fritsche, Jens Kosiol, Andy Schürr, and Gabriele Taentzer. 2018. Short-Cut Rules - Sequential Composition of Rules Avoiding Unnecessary Deletions. In *Software Technologies: Applications and Foundations - STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers (Lecture Notes in Computer Science)*, Manuel Mazzara, Iulian Ober, and Gwen Salaün (Eds.), Vol. 11176. Springer, 415–430. https://doi.org/10.1007/978-3-030-04771-9_30

[13] Lars Fritsche, Jens Kosiol, Andy Schürr, and Gabriele Taentzer. 2019. Efficient Model Synchronization by Automatically Constructed Repair Processes. In *Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science)*, Reiner Hähnle and Wil M. P. van der Aalst (Eds.), Vol. 11424. Springer, 116–133. https://doi.org/10.1007/978-3-030-16722-6_7

[14] Lars Fritsche, Jens Kosiol, Andy Schürr, and Gabriele Taentzer. 2020. Avoiding Unnecessary Information Loss: Correct and Efficient Model Synchronization Based on Triple Graph Grammars. *CoRR* abs/2005.14510 (2020). arXiv:2005.14510

[15] Lars Fritsche, Erhan Leblebici, Anthony Anjorin, and Andy Schürr. 2017. A Look-Ahead Strategy for Rule-Based Model Transformations. In *Proceedings of MODELS 2017 Satellite Event: Workshops (ModComp, ME, EXE, COMMitMDE, MRT, MULTI, GEMOC, MoDeVVa, MDETools, FlexMDE, MDEbug), Posters, Doctoral Symposium, Educator Symposium, ACM Student Research Competition, and Tools and Demonstrations co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, USA, September, 17, 2017 (CEUR Workshop Proceedings)*, Loli Burgueño, Jonathan Corley, Nelly Bencomo, Peter J. Clarke, Philippe Collet, Michalis Famelis, Sudipto Ghosh, Martin Gogolla, Joel Greenyer, Esther Guerra, Sahar Kokaly, Alfonso Pierantonio, Julia Rubin, and Davide Di Ruscio (Eds.), Vol. 2019. CEUR-WS.org, 45–53.

[16] Holger Giese and Robert Wagner. 2009. From model transformation to incremental bidirectional model synchronization. *Journal of Software and Systems Modeling* 8, 1 (2009), 21–43. https://doi.org/10.1007/s10270-008-0089-9

[17] Susann Gottmann, Frank Hermann, Nico Nachtigall, Benjamin Braatz, Claudia Ermel, Hartmut Ehrig, and Thomas Engel. 2013. Correctness and Completeness of Generalised Concurrent Model Synchronisation Based on Triple Graph Grammars. In *Proceedings of the Second Workshop on the Analysis of Model Transformations (AMT 2013), Miami, FL, USA, September 29, 2013 (CEUR Workshop Proceedings)*, Benoit Baudry, Jürgen Dingel, Levi Lucio, and Hans Vangheluwe (Eds.), Vol. 1077. CEUR-WS.org.

[18] Reiko Heckel and Gabriele Taentzer. 2020. *Graph Transformation for Software Engineers - With Applications to Model-Based Development and Domain-Specific Language Engineering*. Springer. https://doi.org/10.1007/978-3-030-43916-3

[19] Frank Hermann, Hartmut Ehrig, Claudia Ermel, and Fernando Orejas. 2012. Concurrent Model Synchronization with Conflict Resolution Based on Triple Graph Grammars. In *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science)*, Juan de Lara and Andrea Zisman (Eds.), Vol. 7212. Springer, 178–193. https://doi.org/10.1007/978-3-642-28872-2_13

[20] Frank Hermann, Hartmut Ehrig, Ulrike Golas, and Fernando Orejas. 2010. Efficient analysis and execution of correct and complete model transformations based on triple graph grammars. In *Proceedings of the First International Workshop on Model-Driven Interoperability, MDI@MoDELS 2010, Oslo, Norway, October 3-5, 2010*, Jean Bézivin, Richard Mark Soley, and Antonio Vallecillo (Eds.). ACM, 22–31. https://doi.org/10.1145/1866272.1866277

[21] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, Yingfei Xiong, Susann Gottmann, and Thomas Engel. 2015. Model synchronization based on triple graph grammars: correctness, completeness and invertibility. *Journal of Software and Systems Modeling* 14, 1 (2015), 241–269. https://doi.org/10.1007/s10270-012-0309-1

[22] Thomas Hettel, Michael Lawley, and Kerry Raymond. 2008. Model Synchronisation: Definitions for Round-Trip Engineering. In *Theory and Practice of Model Transformations - 1st International Conference, ICMT@TOOLS 2008, Zurich, Switzerland, July 1-2, 2008, Proceedings (Lecture Notes in Computer Science)*, Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio (Eds.), Vol. 5063. Springer, 31–45. https://doi.org/10.1007/978-3-540-69927-9_3

[23] Timo Kehrer. 2015. *Calculation and propagation of model changes based on user-level edit operations: a foundation for version and variant management in model-driven engineering.* Ph.D. Dissertation. University of Siegen.

[24] Felix Klar, Marius Lauder, Alexander Königs, and Andy Schürr. 2010. Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In *Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*, Gregor Engels, Claus Lewerentz, Wilhelm Schäfer, Andy Schürr, and Bernhard Westfechtel (Eds.). Lecture Notes in Computer Science, Vol. 5765. Springer, 141–174. https://doi.org/10.1007/978-3-642-17322-6_8

[25] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2008. Detecting and Repairing Inconsistencies across Heterogeneous Models. In *First International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008.* IEEE Computer Society, 356–364. https://doi.org/10.1109/ICST.2008.23

[26] Jens Kosiol, Lars Fritsche, Andy Schürr, and Gabriele Taentzer. 2020. Double-pushout-rewriting in S-Cartesian functor categories: Rewriting theory and application to partial triple graphs. *Journal of Logical and Algebraic Methods in Programming* 115 (2020), 100565.

[27] Erhan Leblebici. 2018. *Inter-Model Consistency Checking and Restoration with Triple Graph Grammars.* Ph.D. Dissertation. Darmstadt University of Technology, Germany.

[28] Nuno Macedo and Alcino Cunha. 2016. Least-change bidirectional model transformation with QVT-R and ATL. *Journal of Software and Systems Modeling* 15, 3 (2016), 783–810. https://doi.org/10.1007/s10270-014-0437-x

[29] Fernando Orejas, Artur Boronat, Hartmut Ehrig, Frank Hermann, and Hanna Schölzel. 2013. On Propagation-Based Concurrent Model Synchronization. *Electronic Communication of the European Association of Software Science and Technology* 57 (2013). https://doi.org/10.14279/tuj.eceasst.57.871

[30] Fernando Orejas, Elvira Pino, and Marisa Navarro. 2020. *Incremental Concurrent Model Synchronization using Triple Graph Grammars.* Lecture Notes in Computer Science, Vol. 12076. Springer, 273–293. https://doi.org/10.1007/978-3-030-45234-6_14

[31] Benjamin Pierce, A. Schmitt, and Michael Greenwald. 2003. Bringing harmony to optimism: a synchronization framework for heterogeneous tree-structured data. *Technical Report MS-CIS-03-42* (01 2003).

[32] Andy Schürr. 1994. Specification of Graph Translators with Triple Graph Grammars. In *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16-18, 1994, Proceedings (Lecture Notes in Computer Science)*, Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer (Eds.), Vol. 903. Springer, 151–163. https://doi.org/10.1007/3-540-59071-4_45

[33] Gabriele Taentzer, Manuel Ohrndorf, Yngve Lamo, and Adrian Rutle. 2017. Change-Preserving Model Repair. In *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science)*, Marieke Huisman and Julia Rubin (Eds.), Vol. 10202. Springer, 283–299. https://doi.org/10.1007/978-3-662-54494-5_16

[34] Laurence Tratt. 2008. A change propagating model transformation Language. *Journal of Object Technolology* 7, 3 (2008), 107–124. https://doi.org/10.5381/jot.2008.7.3.a3

[35] Frank Trollmann and Sahin Albayrak. 2017. Decision Points for Non-determinism in Concurrent Model Synchronization with Triple Graph Grammars. In *Theory and Practice of Model Transformation - 10th International Conference, ICMT@STAF 2017, Marburg, Germany, July 17-18, 2017, Proceedings (Lecture Notes in Computer Science)*, Esther Guerra and Mark van den Brand (Eds.), Vol. 10374. Springer, 35–50. https://doi.org/10.1007/978-3-319-61473-1_3

[36] Nils Weidmann, Anthony Anjorin, Lars Fritsche, Gergely Varró, Andy Schürr, and Erhan Leblebici. 2019. Incremental Bidirectional Model Transformation with eMoflon: : IBeX. In *Proceedings of the 8th International Workshop on Bidirectional Transformations co-located with the Philadelphia Logic Week, Bx@PLW 2019, Philadelphia, PA, USA, June 4, 2019 (CEUR Workshop Proceedings)*, James Cheney and Hsiang-Shang Ko (Eds.), Vol. 2355. CEUR-WS.org, 45–55.

[37] Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Song Hui, Hong Mei, Yingfei Xiong, Haiyan Zhao, Zhenjiang Hu, Masato Takeichi, Song Hui, and Hong Mei. 2008. Beanbag: Operation-based Synchronization with IntraRelations. In *Grace Technical Reports, GRACE-TR-2008–04.* National Institute of Informatics.

[38] Yingfei Xiong, Hui Song, Zhenjiang Hu, and Masato Takeichi. 2009. Supporting Parallel Updates with Bidirectional Model Transformations. In *Theory and Practice of Model Transformations - 2nd International Conference, ICMT@TOOLS 2009, Zurich, Switzerland, June 29-30, 2009. Proceedings (Lecture Notes in Computer Science)*, Richard F. Paige (Ed.), Vol. 5563. Springer, 213–228. https://doi.org/10.1007/978-3-642-02408-5_15

[39] Yingfei Xiong, Hui Song, Zhenjiang Hu, and Masato Takeichi. 2013. Synchronizing concurrent model updates based on bidirectional transformation. *Journal of Software and Systems Modeling* 12, 1 (2013), 89–104. https://doi.org/10.1007/s10270-010-0187-3

A Overview over conflicts and actions for propagation

In Table 1 we give an overview of the different possible conflicts depending on the annotation of the DPC.

B Evaluation Measurements

Figure 19 and Fig. 20 depict the evaluation measurements of our second TGG, which is based on Modisco [5] and a custom documentation model. The description for these measurements can be found in Section 7.

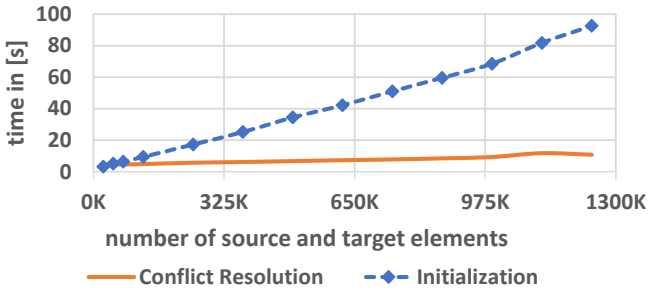

Figure 19. Increasing model size with constant number of 100 conflicts

source \ target	{}	–	/	#	n
{}		pdc	pdc, cpc		cpc
–	pdc		pdc, cpc	pdc	pdc, cpc
/	pdc, cpc	pdc, cpc	pdc, cpc	pdc, cpc	pdc, cpc
#		pdc	pdc, cpc	acc	cpc
n	cpc	pdc, cpc	pdc, cpc	cpc	cpc

Table 1. Overview of potential conflicts depending on the annotations in the DPG where *acc* stands for attribute change, *pdc* for preserve-deletion, and *cpc* for correspondence preservation conflict.

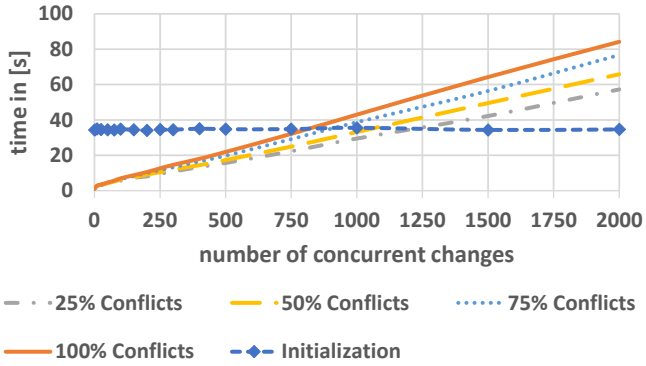

Figure 20. Increasing number of changes with constant model of 500K nodes

C Intermediate conflict resolution steps

Figure 21 depicts the model of our running example after *Local CC*, *Translate* and *Repair* have been applied, while Fig. 22 shows the corresponding delta precedence graph with conflicts $\hat{C}_1$, $\hat{C}_2$ and $\hat{C}_3$. Figure 23 and Fig. 24 depict the final model and delta precedence graph, respectively.

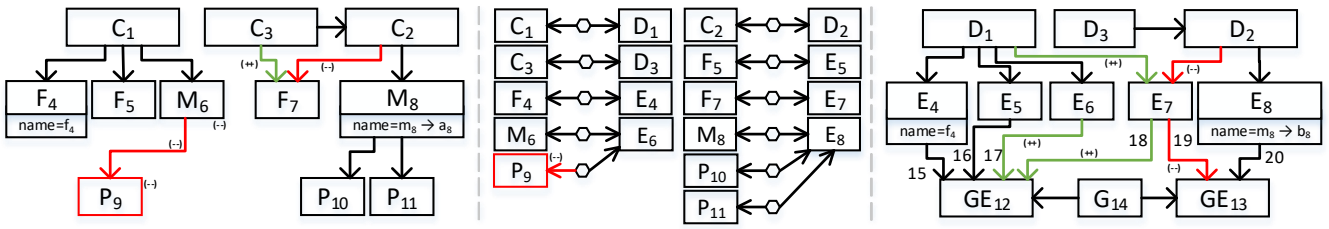

Figure 21. Running Example – Model and Delta after LocalCC, Repair and Translate

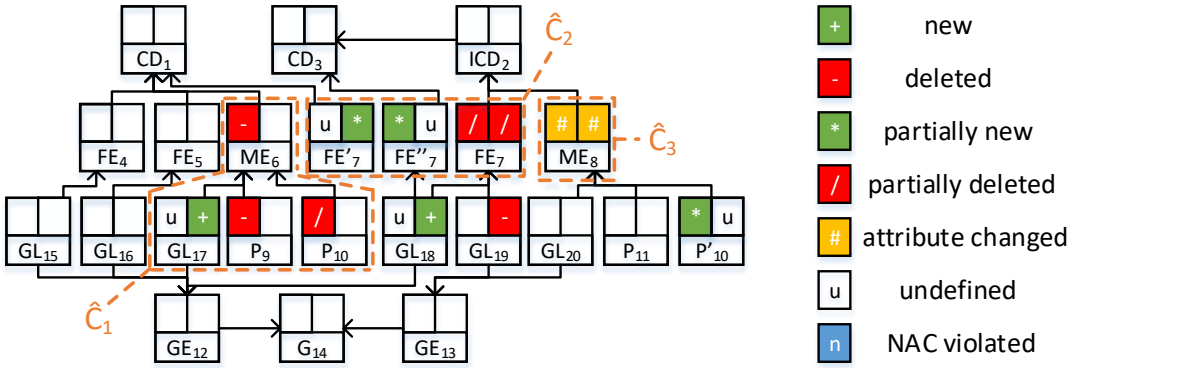

Figure 22. Running Example – Delta Precedence Graph after LocalCC, Repair and Translate

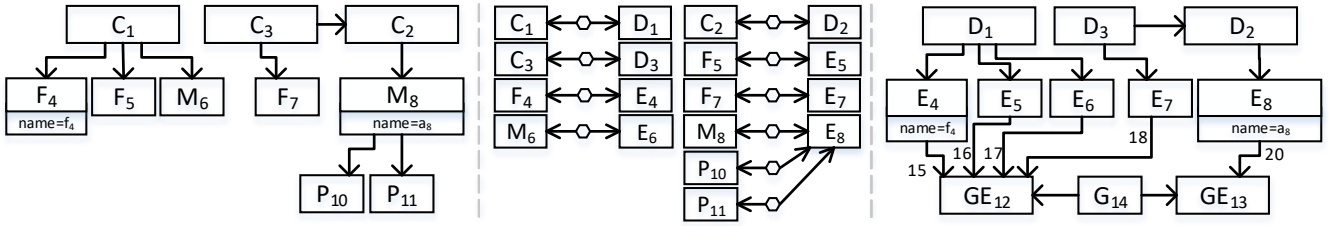

Figure 23. Running Example – Synchronized Model

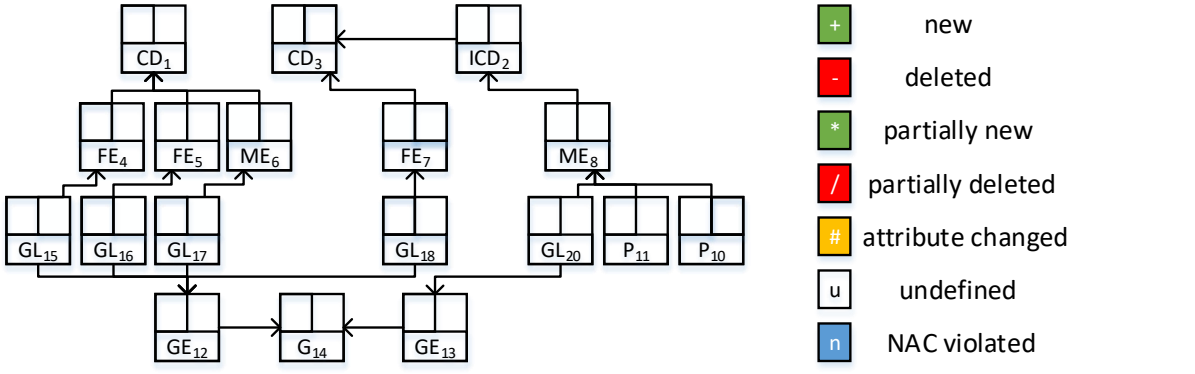

Figure 24. Running Example – Synchronized Precedence Graph