Incremental Concurrent Synchronization of Models with Conflict Resolution*

Elvira Pino
Universitat Politècnica de Catalunya,
Barcelona, Spain
pino@cs.upc.edu

Marisa Navarro
Universidad del País Vasco,
San Sebastián, Spain
marisa.navarro@ehu.es

Fernando Orejas
Universitat Politècnica de Catalunya,
Barcelona, Spain
orejas@cs.upc.edu

In the context of software model-driven development, artifacts are specified by several models describing different aspects, e.g., different views, dynamic behaviour, structure, distributed information, etc. Then, maintaining and repairing consistency of the whole specification became crucial issues if the models can be separately developed and updated. *Model Synchronization* is the process of restoring consistency after the update of one or several of the models. In a previous work, we approached the case of non-concurrent updates and provided an incremental procedure that improves other related proposals. In the present work, we approach the more complex case when conflicts may arise due to concurrently updating different models. Specifically, based on the *Triple Graph Grammar* approach, we propose a two-step algorithm `CSynch`: First, a procedure `CMark` marks the elements in the models which are affected by the concurrent update. As a result we make explicit the possible conflicts and delimit the affected part of the models. In a second stage, a procedure `CRepair` solves conflicts and repairs consistency. Moreover, `CRepair` only works over the marked part of the models, that is, over the part affected by the update, thus incrementality is guaranteed. In addition, we identify and formalize when a synchronizing solution can be considered optimal and state that our procedure `CSynch` is sound and maximizes updating when solving conflicts.

1 Introduction

In the context of model-driven development, artifacts are specified by several models describing its various aspects, e.g., different views, dynamic behaviour, structure, interactions, etc. Moreover, a given set of models is said to be *consistently integrated* if they describe some software artifact. Along the process of designing and implementing an artifact, and also after the artifact is implemented, it is common to modify or update some aspects of a given model, or of several models. These changes may cause an inconsistency between the given set of models. To restore consistency, we have to *propagate* these modifications to the rest of the models. This process is called *model synchronization*. If each time, we propagate just the updates on one model, synchronization is *sequential*, but if we propagate simultaneously updates on several models, synchronization is called *concurrent*. The sequential case is quite simpler than the concurrent one, since in the latter case we have to deal with possible inconsistencies between the modifications applied to different models, implying that in the synchronization process we may need to backtrack some updates. For simplicity, we assume that the given set of models describing an artifact consists just of two models.

A simple but powerful way of describing a class of consistently integrated (synchronized) models is by using a *Triple Graph Grammar* (TGG) [24, 25], consisting of a set of rules on triple graphs. That is, every derivation using the rules of a TGG, generates a triple graph that represent two synchronized

*This work has been partially supported by funds from the Spanish Research Agency (AEI) and the European Union (FEDER funds) under grant GRAMM (ref. TIN2017-86727-C2-1-R and TIN2017-86727-C2-2-R)

Submitted to:
PROLE 2019

models. The TGG approach provides techniques and tools that allow the general formulation and resolution of problems associated to synchronization. In these years the associated techniques have had considerable success, producing a large amount of contributions of proven utility.

In [22], we showed that model synchronization in the sequential case, can be performed incrementally, so that the final consistently integrated models must not be rebuilt from scratch. We called this property *incremental consistency*. Moreover, in that paper, we provided an incremental procedure that improves other related proposals [9, 20, 11] for the general case, when no restrictions are imposed on TGGs, nor on the kind of graphs considered.

In the current paper, we propose a solution for the concurrent case based on similar ideas. We consider that a TGG describes the class of consistent pairs of models (triple models), and we assume that every triple model has an associated derivation d that defines dependency relations between the elements of the models for that derivation. Being specific, given a concurrent update, we propose a two-step algorithm `CSynch` based on the definition and application of *derived rules* from the original grammar rules and the dependencies defined by derivations. In a first stage we assume that updates are traceable by the grammar in the sense that updates should be definable as additions and revocations of derivation steps of the original derivation. Then, a procedure `CMark` uses *derived marking rules* to mark the elements as intended to be added or deleted, required for additions or affected by deletions (+, `x`, ! and ?, respectively), accordingly to the update and the dependency relations. As a result we are able to make explicit the possible conflicts and delimit the affected part of the triple graph. In the second stage, a procedure `CRepair` uses marks to solve possible conflicts and repair the consistency of the models by using *derived re-creating (unmarking) rules*. Our derived rules are defined from the grammar and can be concurrently applied over the models in order to mark or unmark, in contrast to other approaches [34, 35, 13, 10, 31, 32], which are *propagation-based* [21]. Moreover, the procedure only acts over the marked part of the concurrent model, that is, over the part affected by the update, so that, incrementality is guaranteed.

The rest of the paper is organized as follows. In Subsect. 2.1 and 2.3, we summarize the basic and preliminary notions and terminology required in the rest of the paper, while in 2.2 we introduce a running example for the paper. In Sect. 3 we introduce and formalize the properties that should be satisfied by the synchronizing solutions in order to be considered optimal solutions. And, in Sect. 4, we propose a sound synchronizing algorithm which can solve the possible conflicts that may arise from concurrent update, and, in addition satisfies to be incremental and maximize the concurrent update. Finally, in Sect. 5 we present related work, conclude and describe future work.

2 Preliminaries

In this section, we describe some basic notions and terminology concerning, model transformation and model synchronization with Triple Graph Grammars (TGGs). Moreover, we introduce the example that we will use in the paper.

2.1 Basic Definitions: Consistently Integrated Models

We consider that models are some kind of typed graphs with attributes (see, e.g., [5]). This means that our models consist of nodes, edges and attributes, which are values associated to nodes and edges. Moreover, a type graph, which is similar to a metamodel, describes the kind of elements (nodes, edges and attributes) that are part of the given class of models. Second, we consider that integrated models are not just pairs of graphs but *triple graphs* that, in addition, provide a correspondence between elements of the given models. For example, on the left in Fig. 1 we can see the type graph of the example introduced in Subsect. 2.2 and used along the paper. Formally, a triple graph $\overline{G} = (G^S \overset{s_G}{\leftarrow} G^C \overset{t_G}{\rightarrow} G^T)$ consists of a *source graph* G^S and a *target graph* G^T, which are related via a *correspondence graph* G^C and two mappings (graph morphisms) $s_G : G^C \rightarrow G^S$ and $t_G : G^C \rightarrow G^T$ specifying how source elements correspond to target elements. For example, on the left in Fig. 2 we can see a triple graph typed by the graph on the left of Fig. 1. For simplicity, we use the notation $\langle G^S, G^T \rangle$, as an equivalent shorter notation for triple graphs, whenever the explicit correspondence graph can be omitted.

$$\begin{tikzcd}
\overline{L} \arrow[r, hook] \arrow[d, "m"'] & \overline{R} \arrow[d, "m'"] \\
\overline{G}_1 \arrow[r, hook] & \overline{G}_2
\end{tikzcd}$$

A *Triple Graph Grammar* (TGG) $\mathcal{G}$ [24, 25] consists of a start triple graph, $\overline{SG}$[1], and a set of production rules of the form $r : \overline{L} \rightarrow \overline{R}$, where $\overline{L}$ and $\overline{R}$ are triple graphs and $\overline{L} \subseteq \overline{R}$. Then, $\mathcal{L}(\mathcal{G}) = \{\overline{G} \mid \overline{SG} \overset{*}{\Rightarrow} \overline{G}\}$ is called the class of *consistently integrated models* and $\mathcal{D}(\mathcal{G}) = \{\overline{SG} \overset{*}{\Rightarrow} \overline{G}\}$ is the set of derivations defined by $\mathcal{G}$, where $\overset{*}{\Rightarrow}$ is the reflexive and transitive closure of the one step transformation relation $\Rightarrow$ defined as follows: $\overline{G}_1 \Rightarrow \overline{G}_2$ if there is a production rule $r : \overline{L} \rightarrow \overline{R}$ in $\mathcal{G}$ and a matching monomorphism $m : \overline{L} \rightarrow \overline{G}_1$ such that $\overline{G}_2$ can be obtained by replacing (the image of) $\overline{L}$ in $\overline{G}_1$ by (a corresponding image of) $\overline{R}$. Formally, this means that the diagram above on the right is a pushout in the category of triple graphs. In this case, we write $\overline{G}_1 \overset{r,m}{\Rightarrow} \overline{G}_2$, or just $\overline{G}_1 \Rightarrow \overline{G}_2$ if r and m are understood.

2.2 Running Example

In this subsection we introduce the example that is used to illustrate our techniques. It is a simplified, and slightly modified, version of the well-known transformation between class diagrams and relational schemas. The type graphs of source, target and correspondence models are depicted on the left of Fig.1. Source models, whose type graph is depicted on the left, consist of three kinds of nodes: classes, attributes and sub-attributes, and three kinds of edges: An edge between two classes represents a subclass relationship between them; attributes are bound to their associated classes by the second kind of edges as well as sub-attributes to their associated attribute. Similarly, the type graph of target models is depicted on the right of the figure, consisting of tables, columns and sub-columns, together with edges between them. Finally, in the middle, there is the type graph of the correspondence models, consisting of

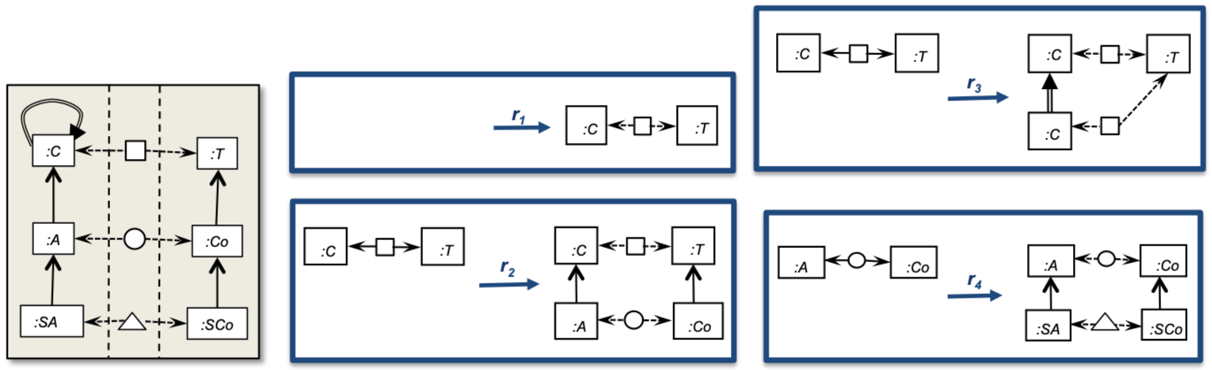


Figure 1: Type graph, four rules for class-to-table transformations

[1] In general, without loss of generality, we will consider that $\overline{SG}$ is always the empty triple graph.

three kinds of nodes: square nodes to bind classes with their associated tables, and round nodes to bind attributes with their associated columns and triangle nodes to bind sub-attributes with their associated sub-columns.

The rules of a TGG defining the transformation between class diagrams and relational schemas are depicted on the right of Fig. 1. Rule r_1, *Class2Table*, creates a new class and its corresponding table, together with the correspondence element that relates the class and the table. Rule r_2, *Attribute2Column*, given a class C and a corresponding table T, creates an attribute A of C, a related column Co of T, and their associated correspondence element. Rule r_3, *Subclass2Table*, given a class C and a corresponding table T, creates a new class C that is a subclass of C. In this case, C is related to T through a new correspondence element. Finally, rule r_4, *SubAttribute2SubColumn*, creates a new sub-attribute together with its corresponding sub-column.

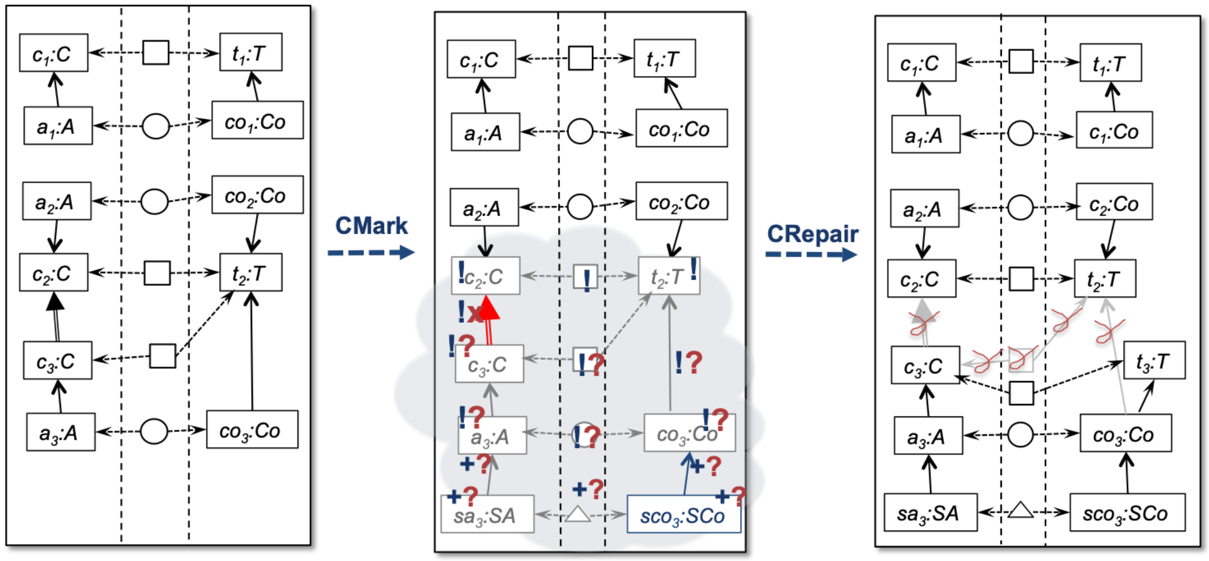


Figure 2: Concurrent update, marked affected area with conflict and possible repair

Given the integrated model on the left of Fig. 2, we want to remove the subclass relation between c_2 and c_3 in the source model, and add a new sub-column sco_3 to the column co_3 in the target model. In the middle of the figure some elements of the triple graph have been marked as intended to be added or deleted ($Matt \subseteq \{+, \mathtt{x}, !, ?\}$) according to the update or some dependency relations that we describe below. Notice that some elements have contradictory marks. This tells us that some conflicting situations would arise if the update was already applied. Finally, on the right of the figure, there is one possible repair of the marked triple graph that avoids conflicts and restores consistency. As we will see, this repair can be made incrementally, acting only on the affected elements (grey area) without having to rebuild the whole triple graph.

2.3 Dependency Relations

Incrementality requires to know what part of the model is affected by the update so that the rest can remain unchanged. For this purpose, we use information from the derivation that generated the given integrated model. In [22], we show that just saving some dependency information associated to the given derivation is enough for our purposes. The idea is to define some dependency relations between the elements (nodes, edges and attributes) of the given integrated model $\overline{G}$ that describe if an element e_1 was needed for the creation of e_2 in a given derivation d. The first relation, called *strict dependency* and denoted $e_1 \lhd_d e_2$, holds if e_1 is matched by the left-hand side of the rule that created e_2. For instance, when deleting the subclass edge between classes c_3 and c_2 in the triple graph on the left of Fig. 2, we have $c_2 \lhd_d c_3$ and $t_2 \lhd_d c_3$, since the application of rule Subclass2Table that creates c_3 has to match its left hand side to c_2 and t_2. The second relation, called *interdependency*, denoted $e_1 \bowtie_d e_2$, holds if e_1 and e_2 are created by the same rule. For instance, in Fig. 2, $c_2 \bowtie_d t_2$, since they are both created by the same application of the *Class2Table* rule in d. Finally, *dependency*, denoted $\unlhd_d$, is the reflexive and transitive closure of the union of $\lhd_d$ and $\bowtie_d$.

Definition 1 (Dependency Relations [22]) *Given $\mathcal{G}$ and a derivation $d : \overline{SG} \overset{*}{\Rightarrow} \overline{G}$, we define the following relations on elements of $\overline{G}$:*

$$\begin{tikzcd}
\overline{L} \arrow[r, hook] \arrow[d, "m"] & \overline{R} \arrow[d, "m'"] \\
\overline{G}_{i-1} \arrow[r, hook] & \overline{G}_i
\end{tikzcd}$$

1. *Strict dependency: $\lhd_d$ is the smallest relation satisfying that if d includes the transformation step depicted, then for every e in $\overline{L}$ and e' in $\overline{R} \setminus \overline{L}$, $m(e) \lhd_d m'(e')$.*
2. *Strict interdependency: $\bowtie_d$ is the smallest relation satisfying that if d includes the transformation step depicted, then for every e, e' in $\overline{R} \setminus \overline{L}$, $m'(e) \bowtie_d m'(e')$.*
3. *Dependency: $\unlhd_d = (\lhd_d \cup \bowtie_d)^*$.*

3 Optimal Synchronizing Solutions for Concurrent Updates

We consider that an *update* or *modification* [7] u on a graph G, is a span of inclusions $u : G \leftarrow K \rightarrow G'$ for some graph K. Intuitively, the elements in G that are not in K are the elements deleted by u, and the elements in G' that are not in K are the elements added by u. So, K consists of all the elements of G that remain invariant after the modification.

In [21], two updates defined on the same model G, $u_1 : G \leftarrow K_1 \rightarrow X_1$ and $u_2 : G \leftarrow K_2 \rightarrow X_2$ are characterized to be *conflict-free*, if we can find graphs K_1' and K_2' such that diagrams (2), (3) and (4) in Fig. 3 are pushouts, where K is the graph build by the pullback diagram (1). In this case, u_1 and u_2 can be merged into an update $u_3 = u_1 \otimes u_2$ as shown on the right diagram in Fig. 3.

$$\begin{tikzcd}
G \arrow[rr, bend left, "u_1"] \arrow[dd, bend right, "u_2"'] & K_1 \arrow[l] \arrow[r] & X_1 \\
K_2 \arrow[u] \arrow[d] \arrow[ur, phantom, "(1)"] & K \arrow[l] \arrow[r] \arrow[u] \arrow[d] \arrow[ur, phantom, "(2)"] \arrow[dl, phantom, "(3)"] & K_1' \arrow[u] \arrow[d] \arrow[dl, phantom, "(4)"] \\
X_2 & K_2' \arrow[l] \arrow[r] & X
\end{tikzcd}
\qquad
\begin{tikzcd}
G \arrow[r, dotted, "u_1"] \arrow[d, dotted, "u_2"'] \arrow[dr, dotted, "u_3"] & X_1 \arrow[d, dotted, "u_2'"] \\
X_2 \arrow[r, dotted, "u_1'"'] & X
\end{tikzcd}$$

Figure 3: Condition for conflict-free updates u_1 and u_2 in a same domain.

$$\begin{tikzcd}
G^S \arrow[r, Leftrightarrow] \arrow[d, dotted, "u^S"] & G^T \arrow[d, dotted, "u^T"] \\
H^S \arrow[r, Leftrightarrow, "?"] & H^T
\end{tikzcd}$$

In the non-concurrent case, given a triple graph $\overline{G}$ and an update $u^S : G^S \leftarrow K^S \rightarrow H^S$ on the source graph, the *synchronization problem* [15] is to find an update $u^T : G^T \leftarrow K^T \rightarrow H^T$ such that $\overline{H}$ is consistently integrated. Solutions are usually based on the so called *forward and backward transformations* which use derived rules from the grammar to build H^T from H^S (respectively, H^S from H^T). In general, there may be no solution to the synchronization problem since such a model H^T may not exist.

In contrast, a concurrent update, noted $\overline{u} = \langle u^S, u^T \rangle : \overline{G} \leftarrow \overline{K} \rightarrow \overline{H}$, is a pair of updates $u^S : G^S \leftarrow K_S \rightarrow H^S$ and $u^T : G^T \leftarrow K^T \rightarrow H^T$, such that H^S and H^T do not necessarily have to be synchronized, that is, $\overline{H} \notin \mathcal{L}(\mathcal{G})$ maybe. To fix terminology, we say that $\overline{u} : \overline{G} \leftarrow \overline{K} \rightarrow \overline{H}$ is *consistency-preserving* if $\overline{G} \in \mathcal{L}(\mathcal{G})$ implies $\overline{H} \in \mathcal{L}(\mathcal{G})$. Let us assume that we have a concurrent update $\overline{u}$ such that u^S and u^T can cause conflicts. However, since these updates modify different models, they do not interfere directly and conflicts are never explicit. For this reason, detecting and solving conflicts become already more difficult than in the non-concurrent case.

In [21] the detection of conflicts for concurrent updates is based on assuming that the framework provides total functions $fPpg$ and $bPpg$ that satisfy suitable properties and propagate the updates from one context to the other, so that the possible conflicts can be detected. These propagation functions are total and, this means that, even in the absence of conflicts, the result of propagation is not required to be consistent. For instance, the deletion of a given element on the source model may be propagated by $fPpg$ to the deletion of some elements of the target model, independently of the consistency of the original and the resulting integrated models. However, for the purpose of this paper, we require the updates to be *traceable by the grammar* in the sense that, if a concurrent update $\overline{u} : \overline{G} \leftarrow \overline{K} \rightarrow \overline{H}$ is traceable by $\mathcal{G}$ then, it means that its application on $\overline{G}$ can be done by adding and/or revoking steps of the given derivation $d = \overline{SG} \overset{*}{\Rightarrow} \overline{G} \in \mathcal{D}(\mathcal{G})$ that generated $\overline{G}$. This means that, if $\overline{u}$ does not cause conflicts then, it has to be consistency-preserving.

Definition 2 (Concurrent Merge of Updates) *Given a TGG $\mathcal{G}$, a consistently integrated model $\overline{G} \in \mathcal{L}(\mathcal{G})$, two consistency-preserving updates on $\overline{G}$, $\overline{v}_1, \overline{v}_2$, such that (v_1^S, v_2^S) and (v_1^T, v_2^T) are both conflict-free in the source part and the target part, respectively, and let $\otimes$ be the operation defined by the diagrams in Fig. 3. Then, the* concurrent merge *of $\overline{v}_1$ and $\overline{v}_2$, is defined as $\overline{v} = \overline{v}_1 \otimes \overline{v}_2 = \langle v_1^S \otimes v_2^S, v_1^T \otimes v_2^T \rangle$, as shown below.*

$$\begin{tikzcd}
 & & G^S \arrow[r, Leftrightarrow] \arrow[dll, dotted, "v_1^S"'] \arrow[drr, dotted, "v_2^S"] \arrow[dd, dotted, "v^S"] & G^T \arrow[dll, dotted, "v_1^T"'] \arrow[drr, dotted, "v_2^T"] \arrow[dd, dotted, "v^T"] & & \\
X_1^S \arrow[r, Leftrightarrow] \arrow[drr, dotted] & X_1^T \arrow[drr, dotted] & & & X_2^S \arrow[r, Leftrightarrow] \arrow[dll, dotted] & X_2^T \arrow[dll, dotted] \\
 & & X^S \arrow[r, Leftrightarrow] & X^T & &
\end{tikzcd}$$

Proposition 1 *If $\overline{v}_1, \overline{v}_2$ are consistency-preserving then so is $\overline{v} = \overline{v}_1 \otimes \overline{v}_2$.*

Now, we can formalize when a concurrent update does not cause conflicts:

Definition 3 (Non-Conflicting Concurrent Updates) *Given a TGG $\mathcal{G}$ and a consistently integrated model $\overline{G} \in \mathcal{L}(\mathcal{G})$, the we say that $\overline{u} : \overline{G} \leftarrow \overline{K} \rightarrow \overline{H}$ is a* non-conflicting concurrent update *if there exists an update $\overline{v} = \overline{v}_1 \otimes \overline{v}_2$ (as in Def. 2), such that $v_1^S = u^S$ and $v_2^T = u^T$.*

In what follows, we formalize when a concurrent update $\overline{v}$ can be considered a solution for a given synchronizing problem $(\overline{G},\overline{u})$ when $\overline{u}$ may be a conflicting concurrent update. The idea is that, given a conflicting concurrent update $\overline{u}$, then solving conflicts can be seen as finding and applying some $\overline{v}$ that avoids making those additions and deletions that produce the conflicts caused by $\overline{u}$. For this purpose, we first recall some notions and fix some notation.

Definition 4 (Subupdates [21]) *Given two updates $v : G \leftarrow K_1 \rightarrow X$ and $w : X \leftarrow K_2 \rightarrow H$, the composition of v and w, is the update $u = w \circ v : G \leftarrow K \rightarrow H$ where the diagram (1) below is a pullback. If diagram (1) is both a pullback and a pushout, then we say that u* decomposes *in v and w and also that v is a* subupdate *of u, noted $v \preceq u$.*

$$\begin{tikzcd} G & K_1 \arrow[l] \arrow[r] & X & K_2 \arrow[l] \arrow[r] & H \\ & & K \arrow[ul] \arrow[ur] \arrow[u, phantom, "(1)"] & & \end{tikzcd}$$

If the above diagram (1) is a pullback, roughly, it means that K is the intersection of K_1 and K_2, i.e. K includes all the elements of G that are neither deleted by v nor by w. Notice that if $v \preceq u$, it means that v adds and deletes less information than u since, accordingly to the properties of the above diagram (1), every element added or deleted by v must be added or deleted by u, respectively. Intuitively, this means that if an update $\overline{u} : \overline{G} \leftarrow \overline{K} \rightarrow \overline{H}$ is causing conflict, then we can apply some non-conflicting $\overline{v}$ that adds and deletes less information than $\overline{u}$ to avoid the conflicts.

Definition 5 (Consistent Synchronizing Solutions) *Given a TGG $\mathcal{G}$, a consistently integrated model $\overline{G} \in \mathcal{L}(\mathcal{G})$ and a concurrent update $\overline{u} : \overline{G} \leftarrow \overline{K} \rightarrow \overline{H}$, we define the set $SynchSol(\overline{G},\overline{u}) = \{\overline{v} \mid \overline{v} = \overline{v}_1 \otimes \overline{v}_2, v_1^S \preceq u^S, v_2^T \preceq u^T\}$, called the* consistent synchronizing solutions *for $\overline{G}$ and $\overline{u}$.*

$SynchSol(\overline{G},\overline{u})$ may contain consistent solutions that are not really intended. For instance, a solution that just backtracks $\overline{u}$ will be a consistent but non reasonable solution. Similarly, a solution that unnecessarily adds arbitrary information not specified by $\overline{u}$ would not be reasonable. To avoid these solutions, given $\overline{v} : \overline{G} \leftarrow \overline{K}_1 \rightarrow \overline{X} \in SynchSol(\overline{G},\overline{u} : \overline{G} \leftarrow \overline{K} \rightarrow \overline{H})$, we define the following requirements[2] on $\overline{v}$:

1. *Maximal Updating*: $\overline{v}$ must contain as many as possible elements added by $\overline{u}$, and, as few as possible elements that are deleted by $\overline{u}$.
2. *Minimal Side Effect*: $\overline{X}$ must contain as many as possible elements from the original model $\overline{G}$ that are not deleted by $\overline{u}$, and, as few as possible elements that are neither in the original model $\overline{G}$ nor added by $\overline{u}$.

For instance, on the right of Fig. 2 we can see an example of a possible consistent solution which is maximal updating and has minimal side-effects. In contrast, in Fig. 4 neither the solution on the left nor the one in the middle are maximal updating, even though both of them are consistent. The solution on the right of Fig. 4 is maximal updating and has minimal side-effects, but it is not comparable with the one in Fig. 2.

Definition 6 (Optimal Synchronizing Solutions) *Given a TGG $\mathcal{G}$, a derivation $d = \overline{SG} \overset{*}{\Rightarrow} \overline{G} \in \mathcal{D}(\mathcal{G})$ and a concurrent update $\overline{u} : \overline{G} \leftarrow \overline{K} \rightarrow \overline{H}$, then, we say that a consistent solution $\overline{v} = \overline{v}_1 \otimes \overline{v}_2 \in SynchSol(\overline{G},\overline{u})$:*

[2]In some sense, they generalize the properties Maximal preservation and Strong soundness defined in [11], respectively.

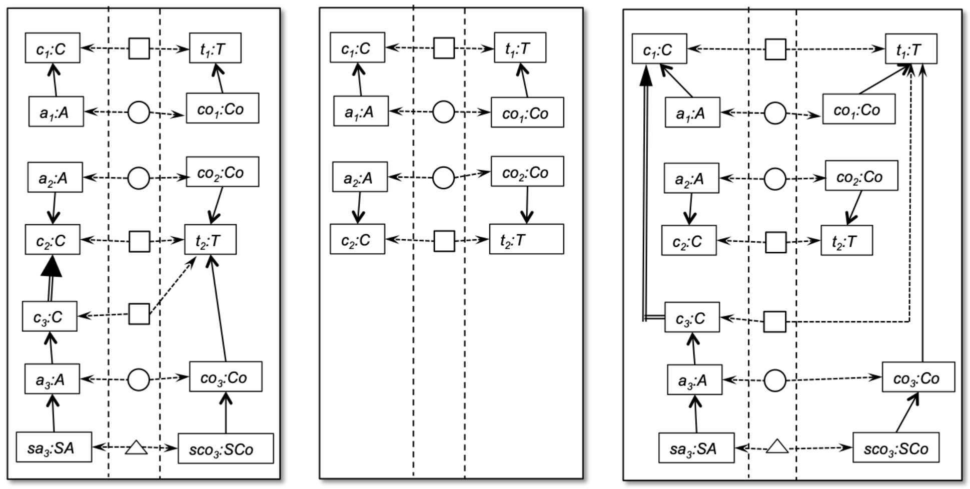


Figure 4: Other three consistent solutions for example in Fig. 2.

1. maximizes updating *if for any other $\overline{w} = \overline{w}_1 \otimes \overline{w}_2 \in SynchSol(\overline{G},\overline{u})$ such that $v_1^S \preceq w_1^S \preceq u^S$ and $v_2^T \preceq w_2^T \preceq u^T$ then, $v_1^S = w_1^S$ and $v_2^T = w_2^T$.*

2. minimizes side-effects *if for any other $\overline{w} : \overline{w}_1 \otimes \overline{w}_2 \in SynchSol(\overline{G},\overline{u})$ such that $v_1^S = w_1^S$ and $v_2^T = w_2^T$ then, $w_1^T \preceq v_1^T$ and $w_2^S \preceq v_2^S$ implies $\overline{v} = \overline{w}$.*

Let us denote as $MaxSynchSol(\overline{G},\overline{u})$ the set of synchronizing solutions that are maximal updating. Let us denote as $OptSynchSol(\overline{G},\overline{u})$ the set of synchronizing solutions that are maximal updating and have minimal side-effects.

4 An Incremental Procedure `CSynch`

In this section we propose a two-step algorithm `CSynch`, based on the definition and application of some *derived rules* from the original grammar rules, for each concurrent update $\overline{u} : \overline{G} \leftarrow \overline{K} \rightarrow \overline{H}$, where u^S and u^T may be in conflict. Our proposal for `CSynch` is not based on propagation but on using concurrent marking rules derived from the grammar, identifying how the elements are affected by the update and, then, concurrently solving possible conflicts and restoring consistency. The marking algorithm `CMark` decorates every element in $\overline{G}$ with a *marking attribute* consisting on a set of marks, $Matt \subseteq \{+, \mathtt{x}, !, ?\}$, meaning added, deleted, required for an addition, or affected by a deletion, respectively. Once the model is marked, an algorithm `CRepair` (non-deterministically) solves conflicts and repairs the model. This process removes *Matt* of every element that is checked as really consistent in the model, so that, at the end of the process, only unmarked element should be considered as effectively created in the solution. Finally, elements that have not been marked by `CMark`, correspond to elements not affected by the update, as a consequence, we can guarantee incrementality since those elements will not be processed by `CRepair` in the unmarking process.

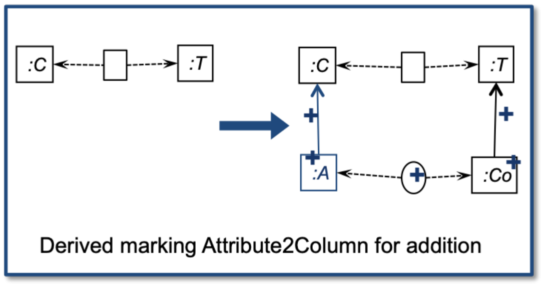
4.1 Marking and Conflict Detecting

Given a grammar $\mathcal{G}$ and a triple graph $\overline{G}$, we want to extend it to obtain a consistent graph. A straightforward approach would be to use the rules in $\mathcal{G}$ to find a graph $\overline{G}'$ that extends $\overline{G}$. But we can modify the rules in $\mathcal{G}$, so that, if an old rule created an element already in $\overline{G}$, the new rule would just mark it; but if the old rule created a new element not in $\overline{G}$, the new rule would also create it. Roughly, these new rules *reuse* the elements in $\overline{G}$ [11, 14]. For instance, in Fig. 5 we have some examples of derived marking rules used for adding. In [22] these ideas are used for the non-concurrent case but they can be generalized to define a more sophisticated notion of *marking rules with reuse*:

Definition 7 (Derived Marking Rules for Adding) *Given a grammar rule $r : \overline{L} \to \overline{R}$, we say that $r' : \overline{L'} \to \overline{R'}$ is a* derived marking rule *from r if*

1. $\overline{L} \subseteq RemAttr(\overline{L'}) \subseteq \overline{R}$
2. $RemAttr(\overline{R'}) = \overline{R}$

where $RemAttr(\overline{G})$ denotes the graph resulting from removing from $\overline{G}$ the marking attribute Matt. We say that r' is used for adding *if its application adds $+$ to the marking attribute Matt of every element in $\overline{R'}$ not in $\overline{L}$.*



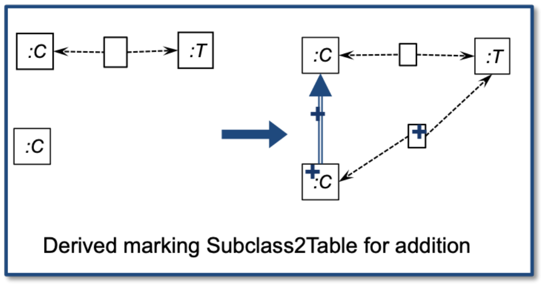


Figure 5: Examples of derived marking rules

Now we can introduce the marking algorithm `CMark` whose goal is to decorate every element in $\overline{G}$ with a set of marking attributes $Matt \subseteq \{+, \mathtt{x}, !, ?\}$. Initially, these sets are empty. Then, derived marking rules are used to add $+$ to every element intended to be added by the update. Second, the algorithm adds $\mathtt{x}$ to every element that has to be revoked as a consequence of an addition. At this point of the process, the original derivation d has to be modified according to the update so that the updated dependency relation can be used to add a mark $!$ to the elements that depend on the elements with a mark $+$. Then, it can mark with $\mathtt{x}$ every element which is intended to be deleted by the update. Finally, it marks as $?$ the elements that depend on elements with a mark $\mathtt{x}$. At the end, every element in the model that is affected by the update, will have a non empty list of marks.

Algorithm 1 (`CMark` **Algorithm**) *Consider that every element in the model is decorated with an attribute $Matt \subseteq \{+, !, \mathtt{x}, ?\}$.*

- *Initialize each element in the model with the empty attribute $Matt = \{\}$.*
- *Compute the dependency relation.*
- *Given a concurrent update, proceed as follows:*
  1. **Addition***: Apply a marking rule for adding every element intended to be added.*

2. **Revocation***: Add* x *to the attribute Matt of every element which is strictly interdependant with a* +*-marked element.*
3. **Addition of the new dependencies***: Add the new dependencies and interdependencies defined by the (original) rules used in 1. to the dependency relation.*
4. **Affected as required***: Add* ! *to the attribute Matt of every element which a* +*-marked element strictly depends from.*
5. **Deletion***: Add* x *to the attribute Matt of every element intended to be deleted.*
6. **Affected as maybe-deleted***: Add* ? *to the attribute Matt of every element that is strictly dependant of a* x*-marked element or strictly interdependant with it.*

For instance, in the middle of Fig. 2 and Fig. 6 we can see examples of a marked model following the above algorithm. In the case of the example in Fig. 6, the concurrent update would consist on adding a subclass relation between classes c_1 and c_2, in the source; and, adding a new sub-column sco_3 to the column co_2 in the target. Again, in the model of the middle, some elements are marked with contradictory marks. Notice that now, possible conflicts arise because of trying to integrate concurrent additions. In fact, this example serves to illustrate that some additions may imply the deletion of elements created by the original derivation, for instance, it is the case of the table t_2. That is, some additions may imply revocation of original derivation steps.

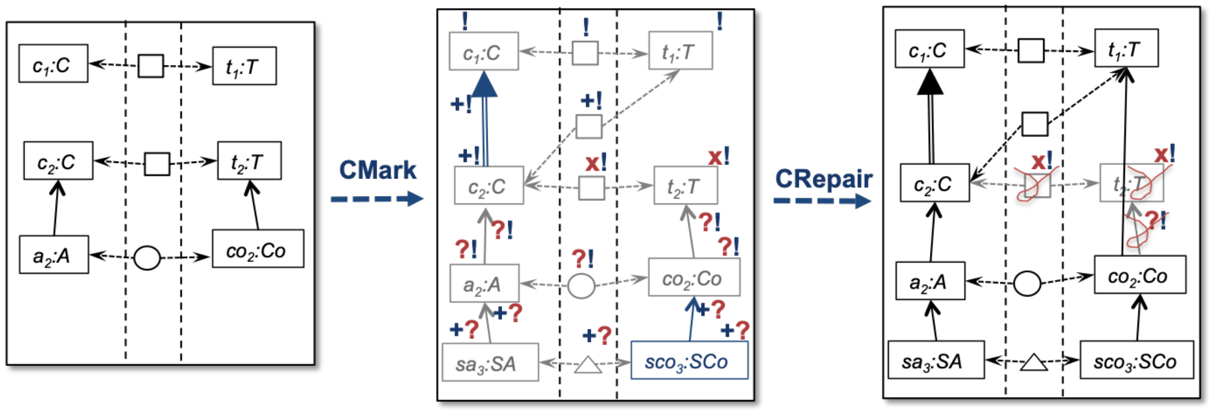


Figure 6: Other example of concurrent updated, marked and possible repair.

4.2 Reparing and Conflict-Solving

Again, following the ideas introduced in [14], now we generalize the notion of translation rules to a notion of *derived recreating concurrent rules* that reuse arbitrary parts of the given source and target graphs, as part of our synchronization algorithm. The idea of the application of rules of this kind is that, on the one hand, it checks if the given graph includes the reusable elements with the right marks in its marking attribute and, on the other, it removes the marking attribute of the recreated elements. Formally, we define:

Definition 8 (Derived Recreating Rules) *Given a rule* $r : \overline{L} \to \overline{R}$*, we say that* $r' : \overline{L'} \to \overline{R}$ *is a* derived recreating rule *from* r *if* $\overline{L} \subseteq RemAttr(\overline{L'}) \subseteq \overline{R}$*, such that*

1. *The elements in* $\overline{L'}$ *from* $\overline{L}$ *must be matched to elements without marks.*
2. *The elements in* $\overline{L'}$ *not in* $\overline{L}$ *can be matched to elements with any mark.*

Notice that the application of a derived recreating rule removes the marking attributes of every element. That is, if a derived recreating rule is applied then elements in $\overline{L'}$ will be recreated. For instance, in Fig. 7 we can see some examples of derived recreating rules. Then, as we pointed out before, using this kind of

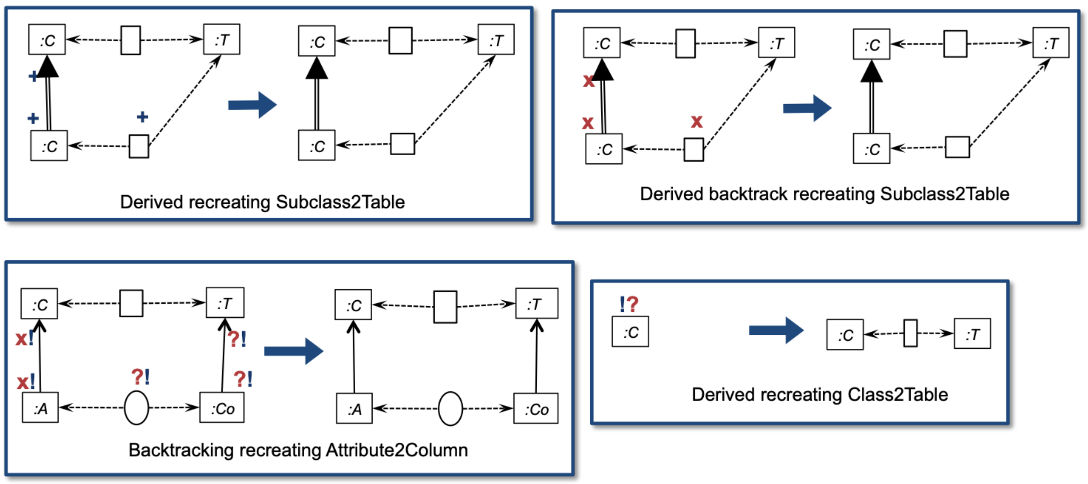

Derived recreating Subclass2Table
Derived backtrack recreating Subclass2Table
Backtracking recreating Attribute2Column
Derived recreating Class2Table
:C
:T
:A
:Co
+
x
?!
x!
!?

Figure 7: Examples of derived recreating rules

(parallel) rules instead of propagations as in [21], allow us to detect conflicting situations. Specifically, we can identify conflicts depending on the values of marks of matched elements. In particular, if $\overline{L'}$ is mapped into elements marked as x then, the application of the rule will backtrack the deletion of those elements. We say it is a *backtracking recreation*. On the contrary, given an element e which is marked as $+$, if no recreating rule is applied such that $e \in \overline{L'}$ then the addition of e will not be really done.

Given a concurrent update, a derivation d and its associated dependency relation, after applying the marking algorithm over the model, the synchronization process can be described as follows: After a concurrent update the applications of rules causing conflict can be avoided. This can be detected because the marking algorithm did not mark the elements added by the update as definitely created but just as $+$, and the deleted element just as x. If a conflict between deleted and added elements arises it can be decided (deterministically or not) if the rule is applied or not. Applying the rule would mean to backtrack the deletion and do the addition. On the contrary, if an added element has a non empty marking attribute (i.e. it is still marked) at the end of the process, this would mean that no rule has been used to create it. Then, eliminating this element would mean to backtrack the addition update. Moreover, since the elements affected by a deletion were marked as ?, again, only at the end of the process of synchronization, it can be decided if these dependant elements could be really recreated or not. If they remain as ?, this means that no rule would have recreated them so they can be removed definitely.

Algorithm 2 (`CSynch` **Algorithm**)

- *Apply* `CMark` *to mark the affected area in the model.*
- *Apply the following algorithm* `CRepair`*:*
  1. **Recreating and conflict solving***: For every element marked as* + *and/or* !,
     - *If possible, apply a non-backtracking recreation.*
     - *If not, decide to apply or not a backtracking recreation.*
  2. **Reusing***: For every* ? *element, apply (if it is possible) a non-backtracking recreation.*
  3. **Removing***: Remove every element with a non-empty set of marks.*

That is, in step 1. of `CRepair`, we first try to recreate every + or ! element without having to backtrack the intended deletions. If it is not possible, this means a conflict has been detected and then we have to (perhaps indeterministically) decide if we priorize addition or deletion. In step 2. we can try to reuse some elements affected by deletions if we do not want to loose its information. However, in this case, backtracking of deletion has to be avoided in order to prevent non maximal-updating solutions. Finally, in step 3. all elements still marked are removed.

Theorem 1 (Soundness and Strong Soundness. Incrementality) `CSynch` *is sound, that is, for every* $\overline{G}' \in \texttt{CSynch}(d, \overline{u})$*, there exists* $\overline{u}' \in SynchSol(\overline{G}, \overline{u})$ *such that* $\overline{u}' : \overline{G} \leftarrow \overline{K} \rightarrow \overline{G}'$*. Moreover,* `CSynch` *is strongly sound, that is,* $\overline{u}' \in MaxSynchSol(\overline{G}, \overline{u})$*. In addition,* `CSynch` *is* incremental*, that is, if* $\overline{G}' \in \texttt{CSynch}(d, \overline{u})$*, such that* $d = \overline{SG} \overset{*}{\Rightarrow} \overline{H} \overset{*}{\Rightarrow} \overline{G}$ [3] *and no element in* $\overline{H}$ *is marked as* x *by* `CMark`*, then* $\overline{G}' \in \texttt{CSynch}(\overline{H} \overset{*}{\Rightarrow} \overline{G}, \overline{u})$*.*

However, `CSynch` can build solutions which does not minimize side-effects. For instance this would be the case if the grammar contains different rules that can be used for the same recreation, such that they differ in the amount of consistent information they create. Avoiding non minimal side-effect solutions could be done by defining a notion of minimal recreation rule which use would be priorized.

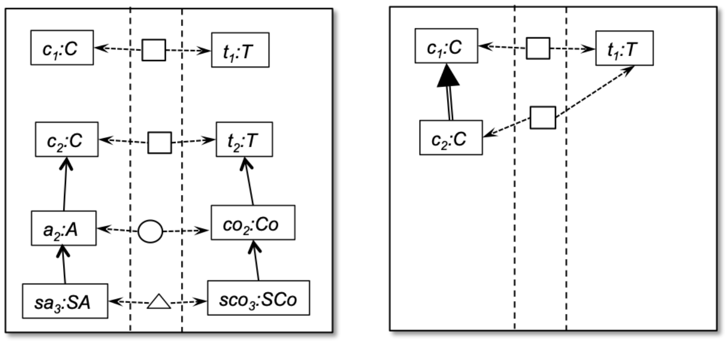


Figure 8: Non maximal-updating solutions for example in Fig. 6.

Consider the solutions on the right of Fig. 2 and Fig. 6. They can be built by `CSynch` and they are maximal updating and have minimal side effect. However, other consistent but non maximal-updating

[3]In fact, it should be if d is permutation equivalent to some $d_0 = \overline{SG} \overset{*}{\Rightarrow} \overline{H} \overset{*}{\Rightarrow} \overline{G}$.

solutions are not computed by `CSynch`. For instance, in the example of Fig. 2, we could backtrack the subclass deletion by using a recreating rule producing the synchronized model on the left of Fig. 4. Similarly, in the case of Fig. 6, if the first recreating rule *Derived recreating Subclass2Table* in Fig. 7 was never applied, at the end of the process the addition of the subclass relation will be backtracked and a different solution, as for instance the one on the left of Fig. 8, could be obtained. None of solutions in Fig. 8 are computed by `CSynch`.

5 Related and Future Work

To our knowledge, the only works addressing the general problem[4] of concurrent synchronization are [35, 13, 10, 21, 31, 32]. All these approaches are propagation-based, which means that synchronization is performed, first, propagating the updates in one model to the other model, then checking if there is any conflict between the propagated updates and the ones previously applied to that model and, if there are, solving the conflicts in some given way, and finally, propagating back the updates in the second model to the first one. That is, defining concurrent synchronization somewhat sequentially. In all cases it is shown that the result obtained is correct. However, as discussed in [21], the results may be not satisfactory enough. For, instance, in [13, 10, 21, 31, 32] synchronization could just undo all the updates, leading to the original models. Only [35] excludes this case but, in that approach, the synchronization procedure may be unable to find correct results, even if they exist. The reason is that they do not consider properties like maximizing updating or producing minimal side-effects, defined in our paper. Moreover, none of these approaches are incremental.

Our approach to incrementality is based on the ideas presented in [22], for the sequential synchronization case. Other approaches based on TGGs that propose incremental solutions to sequential model synchronization are [9, 15, 11, 20] (and some variations on them) but all of them are, in our opinion, not completely satisfactory. In particular, even if the construction of the solution does not start from scratch but from the given integrated model $\overline{G}$, the approaches in [15, 11, 20] have to analyze $\overline{G}$ (for instance, to know what parts of $\overline{G}$ must be modified) so their cost depends on the size of the given model. This is not the case of [9], but their approach only works for the case when source and target models are bijective, which excludes the case where source models are views of target models (or vice versa). In addition, in [9, 15, 20] there may be information loss. As in [15], we use translation attributes. As in [11], we use the same ideas to reuse the information included in the given models but expressed in a formal way. Finally, as in [20], our approach is based on the use of precedence relations. However, their relation is coarser than ours. The reason is that our relations are directly based on a given derivation while in [20], their relation is based on the dependences established by the rules of the TGG. In particular, this means that two given elements of a model may be independent, but their relation may say that one depends on the other.

As future work, we plan to extend our results to the multimodel case, i.e. when synchronizing more than two models. This case has specific complications, see, for instance [29, 3]. It has already been approached in [31, 32], but just as a straightforward generalization of [13], which means that it shares its limitations.

[4]There is some work considering this problem in a more restrictive setting. For instance, in [23] models are restricted to tree-like structures and the target model is an abstract view of the source; and in [33] updates must be defined in terms of a given set of operations.

References

[1] Umeshwar Dayal & Philip A. Bernstein (1982): *On the Correct Translation of Update Operations on Relational Views*. ACM Trans. Database Syst. 7(3), pp. 381–416.

[2] Zinovy Diskin (2011): *Model Synchronization: Mappings, Tiles, and Categories*. In: Generative and Transformational Techniques in Software Engineering III, 6491, Springer, pp. 92–165.

[3] Zinovy Diskin, Harald König & Mark Lawford (2018): *Multiple Model Synchronization with Multiary Delta Lenses*. In: Fundamental Approaches to Software Engineering, 21st International Conference, FASE 2018, Lecture Notes in Computer Science 10802, Springer, pp. 21–37.

[4] Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann & Fernando Orejas (2011): *From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case*. In: Model Driven Engineering Languages and Systems, MODELS 2011, Lecture Notes in Computer Science 6981, Springer, pp. 304–318.

[5] H. Ehrig, K. Ehrig, U. Prange & G. Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs of Theoretical Comp. Sc., Springer.

[6] Hartmut Ehrig, Karsten Ehrig & Frank Hermann (2008): *From Model Transformation to Model Integration based on the Algebraic Approach to Triple Graph Grammars*. ECEASST 10.

[7] Hartmut Ehrig, Claudia Ermel & Gabriele Taentzer (2011): *A Formal Resolution Strategy for Operation-Based Conflicts in Model Versioning Using Graph Modifications*. In: FASE 2011, Lecture Notes in Computer Science 6603, Springer, pp. 202–216.

[8] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa & Wang Chiew Tan (2008): *Quasi-inverses of schema mappings*. ACM Trans. Database Syst. 33(2).

[9] Holger Giese & Robert Wagner (2009): *From model transformation to incremental bidirectional model synchronization*. Software and System Modeling 8(1), pp. 21–43.

[10] Susann Gottmann, Frank Hermann, Nico Nachtigall, Benjamin Braatz, Claudia Ermel, Hartmut Ehrig & Thomas Engel (2013): *Correctness and Completeness of Generalised Concurrent Model Synchronisation Based on Triple Graph Grammars*. In: AMT@MoDELS, Lecture Notes in Computer Science 1077, Springer.

[11] Joel Greenyer, Sebastian Pook & Jan Rieke (2011): *Preventing Information Loss in Incremental Model Synchronization by Reusing Elements*. In: ECMFA 2011, Lecture Notes in Computer Science 6698, Springer, pp. 144–159.

[12] David Hearnden, Michael Lawley & Kerry Raymond (2006): *Incremental Model Transformation for the Evolution of Model-Driven Systems*. In: MoDELS 2006, Lecture Notes in Computer Science 4199, Springer, pp. 321–335.

[13] Frank Hermann, Hartmut Ehrig, Claudia Ermel & Fernando Orejas (2012): *Concurrent Model Synchronization with Conflict Resolution Based on Triple Graph Grammars*. In: FASE 2012, Lecture Notes in Computer Science 7212, Springer, pp. 178–193.

[14] Frank Hermann, Hartmut Ehrig, Ulrike Golas & Fernando Orejas (2014): *Formal Analysis of Model Transformations based on Triple Graph Grammars*. Math. Struct. in Comp. Sc. 24.

[15] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin & Yingfei Xiong (2011): *Correctness of Model Synchronization Based on Triple Graph Grammars*. In: MODELS 2011, Lecture Notes in Computer Science 6981, Springer, pp. 668–682.

[16] Frank Hermann, Hartmut Ehrig, Fernando Orejas & Ulrike Golas (2010): *Formal Analysis of Functional Behaviour for Model Transformations Based on Triple Graph Grammars*. In: ICGT 2010, Lecture Notes in Computer Science 6372, Springer, pp. 155–170.

[17] Martin Hofmann, Benjamin C. Pierce & Daniel Wagner (2011): *Symmetric lenses*. In: POPL 2011, ACM, pp. 371–384.

[18] Martin Hofmann, Benjamin C. Pierce & Daniel Wagner (2012): *Edit lenses*. In John Field & Michael Hicks, editors: *POPL'12*, ACM, pp. 495–508. Available at `http://dl.acm.org/citation.cfm?id=2103656`.

[19] S. Lack & P. Sobocinski (2005): *Adhesive and quasiadhesive categories*. *Theor. Inf. App.* 39, pp. 511–545.

[20] Marius Lauder, Anthony Anjorin, Gergely Varró & Andy Schürr (2012): *Efficient Model Synchronization with Precedence Triple Graph Grammars*. In: *ICGT 2012*, *Lecture Notes in Computer Science* 7562, Springer, pp. 401–415.

[21] Fernando Orejas, Artut Boronat, Hartmut Ehrig, Frank Hermann & Hanna Schölzel (2013): *On Propagation-Based Concurrent Model Synchronization*. In: *BX 2013*, *Electronic Communications of the EASST* 57, European Association of Software Science and Technology, pp. 1–20.

[22] Fernando Orejas & Elvira Pino (2014): *Correctness of Incremental Model Synchronization with Triple Graph Grammars*. In: *ICMT 2014*, *Lecture Notes in Computer Science* 8568, Springer, pp. 74–90.

[23] Benjamin C. Pierce (2005): *Harmony: The Art of Reconciliation*. In: *TGC 2005*, *Lecture Notes in Computer Science* 3705, Springer, p. 1.

[24] Andy Schürr (1994): *Specification of Graph Translators with Triple Graph Grammars*. In: *WG '94*, *Lecture Notes in Computer Science* 903, Springer, pp. 151–163.

[25] Andy Schürr & Felix Klar (2008): *15 Years of Triple Graph Grammars*. In: *ICGT 2008*, pp. 411–425.

[26] Perdita Stevens (2008): *Towards an Algebraic Theory of Bidirectional Transformations*. In: *ICGT'08*, *Lecture Notes in Computer Science* 5214, Springer, pp. 1–17.

[27] Perdita Stevens (2010): *Bidirectional model transformations in QVT: semantic issues and open questions*. *Software and System Modeling* 9(1), pp. 7–20.

[28] Perdita Stevens (2012): *Observations relating to the equivalences induced on model sets by bidirectional transformations*. *ECEASST* 49.

[29] Perdita Stevens (2018): *Towards sound, optimal, and flexible building from megamodels*. In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018*, ACM, pp. 301–311.

[30] James F. Terwilliger, Anthony Cleve & Carlo Curino (2012): *How Clean Is Your Sandbox? - Towards a Unified Theoretical Framework for Incremental Bidirectional Transformations*. In: *ICMT 2012*, *Lecture Notes in Computer Science* 7307, Springer, pp. 1–23.

[31] Frank Trollmann & Sahin Albayrak (2015): *Extending Model to Model Transformation Results from Triple Graph Grammars to Multiple Models*. In: *ICMT 2015*, *Lecture Notes in Computer Science* 9152, Springer, pp. 214–229.

[32] Frank Trollmann & Sahin Albayrak (2017): *Decision Points for Non-determinism in Concurrent Model Synchronization with Triple Graph Grammars*. In: *ICMT 2017*, *Lecture Notes in Computer Science* 10374, Springer, pp. 35–50.

[33] Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Hui Song, Masato Takeichi & Hong Mei (2009): *Supporting automatic model inconsistency fixing*. In: *ESEC/FSE 2009*, pp. 315–324.

[34] Yingfei Xiong, Hui Song, Zhenjiang Hu & Masato Takeichi (2009): *Supporting Parallel Updates with Bidirectional Model Transformations*. In: *ICMT 2009*, *Lecture Notes in Computer Science* 5563, Springer, pp. 213–228.

[35] Yingfei Xiong, Hui Song, Zhenjiang Hu & Masato Takeichi (2013): *Synchronizing concurrent model updates based on bidirectional transformation*. *Software and System Modeling* 12, pp. 89–104.