Electronic Communications of the EASST
Volume 73 (2016)


# Graph Computation Models
# Selected Revised Papers from GCM 2015

## 20 Years of Triple Graph Grammars: A Roadmap for Future Research

Anthony Anjorin, Erhan Leblebici and Andy Schürr

20 pages

Guest Editors: Detlef Plump
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

# 20 Years of Triple Graph Grammars: A Roadmap for Future Research

**Anthony Anjorin[1], Erhan Leblebici[2] and Andy Schürr[2]**

[1]anjorin@chalmers.se
Chalmers | University of Gothenburg

[2]surname@tu-darmstadt.de
Technische Universität Darmstadt

**Abstract:** Triple graph grammars (TGGs) provide a declarative, rule-based means of specifying binary consistency relationships between different types of graphs. Over the last 20 years, TGGs have been applied successfully in a range of application scenarios including: model generation, conformance testing, bidirectional model transformation, and incremental model synchronisation.

In this paper, we review the progress made in TGG research up until now by exploring multiple research dimensions, including both the current frontiers of TGG research as well as important future challenges. Our aim is to provide a roadmap for the coming years of TGG research by stating clearly what we regard as adequately researched, and what we view as still unexplored potential.

**Keywords:** Triple Graph Grammars, Bidirectional Model Transformation, Model Synchronisation

## 1 Introduction and motivation

Triple graph grammars (TGGs) [Sch94] provide a declarative, rule-based means of specifying binary consistency relationships between different types of graphs. Over the last 20 years, TGGs have been primarily used as a *bidirectional* model transformation language. Models are encoded as graphs and model transformations, used to maintain the consistency of related pairs of models in a concurrent engineering scenario by propagating changes in both directions, are derived from a given TGG specification. Bidirectional transformations (*bx*) are highly relevant in numerous domains [CFH+09], and multiple frameworks, approaches, and tools have been suggested for *bx* (cf. [Ste08] for an overview). Important application scenarios of *bx* include model synchronisation, round-tripping, and realising editable views [DWGC14].

As a *bx* approach, TGGs enjoy both a solid formal foundation rooted in *algebraic graph transformation* [EEPT06], as well as rich and varied tool support (cf. [LAS+14b, HLG+13] for an overview). Research on TGGs has been fairly active in recent years, and results, foci, and challenges have shifted and developed considerably since the last visionary paper on TGGs [SK08].

This paper provides essentially an update of [SK08], reviewing and reporting on results and progress after 20 years of TGG research. We extend the challenges posed in [SK08] to cover five dimensions, with most extensions strongly influenced by recent research in the field of *bx* in

general. In each dimension we (i) review the relevant *status quo*, providing ample references to existing formalisations and implementations, and (ii) pose future challenges required to push the current frontiers of TGG research in what we believe are the most promising directions.

The rest of the paper is structured as follows: Section 2 provides a brief introduction to TGGs with a running example taken from the *bx* example repository [CMSG14]. Section 3 presents our main contribution as a radar chart of research dimensions, motivated and clarified informally using the running example. In Section 4 we compare our contribution with other related "visionary" papers, and conclude in Section 5 with a brief summary and outlook on future work.

## 2 Running Example and Preliminaries

A TGG specification consists of a *TGG schema* and a set of *TGG rules*. A TGG schema is a triple of source, correspondence, and target metamodels, which define the languages of models of the two domains (source and target) over which consistency is to be specified, as well as a language of explicit correspondence links that are created between related elements of source and target models. TGGs are symmetric as the consistency relation is not in any way directed. Referring to one domain as "source" and the other as "target" is, therefore, just a historic convention.

The TGG schema for our running example, a variant of *ModelTests* [Ste] from the *bx* examples repository [CMSG14], is depicted in Fig. 1. The source metamodel represents a simple language of class diagrams consisting of a hierarchy of diagrams (`Diagram`), classes (`Class`), and methods (`Method`). All elements are named (string attribute `name`), and methods can be public or private (boolean attribute `public`). As multiplicities can only express simple constraints (diagrams/classes contain arbitrarily many classes/methods, and every class/method is connected to exactly one diagram/class), an additional constraint language is required to formulate more complex domain constraints. In this paper, we use so called *graph conditions* (cf., e.g., [RAB+15]), which are less expressive than the Object Constraint Language (OCL) but fit better in the general framework of algebraic graph transformation and consequently TGGs (cf., e.g., [AST12]). Two domain constraints, `nc` for *negative constraint* and `pc` for *positive constraint*, are used in the source metamodel to ensure that no two classes exist with the same name, and that every class has a public method, respectively.

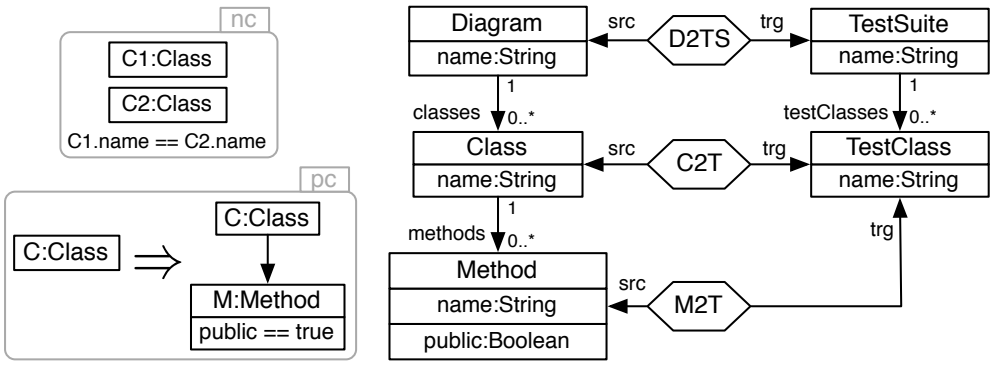


Figure 1: TGG schema: source, correspondence, and target metamodels with domain constraints

A negative constraint is a pattern $N$ that must not exist in a valid model. A positive constraint is of the form $P \to C$, where $P$ and $C$ are patterns representing the premise and conclusion of the constraint, respectively. If the premise can be found in a model then the conclusion, as an extension of the premise, must also hold. For formal definitions of graph conditions and multiplicities we refer to [Tae12, RAB+15]. To clearly differentiate patterns from (meta)models, patterns are placed in a light grey border that is tagged with the name of the pattern, and capital letters are used for the names of objects in patterns. Where there is only one possible association between types, the name of the link is omitted to prevent diagram clutter (e.g., the link between `C:Class` and `M:Method` can only be of type `methods` and is thus omitted).

Patterns contain *attribute conditions* such as `C1.name == C2.name` in `nc`, or `public == true` in `pc`, where the latter condition is inlined in the object `M:Method`. The target metamodel consists of test suites (`TestSuite`) with test classes (`TestClass`), while the correspondence metamodel provides types for "links" between source and target elements: diagrams correspond to test suites, classes and methods correspond to test classes.

The consistency relation between class diagrams and test suites is specified informally in [Ste], and we extend it by introducing an additional requirement concerning methods (condition 3 below). To fit in a TGG context, the conditions are additionally interpreted as requirements on the correspondence model between consistent source and target models:

1. For every diagram there exists a test suite with the same name.
2. For every class with name "Foo", there exists at least one test class called "TestFooX" for some natural number $X$.
3. Every public method of a class is associated with all test classes for the class.

An example of a consistent triple of source, correspondence, and target models is depicted in Fig. 2. The source model is a diagram with two classes, while the target model is a test suite with three test classes. Note that every class has at least one public method (cf. source domain constraint `pc`), and every public method is connected to every test class for its containing class.

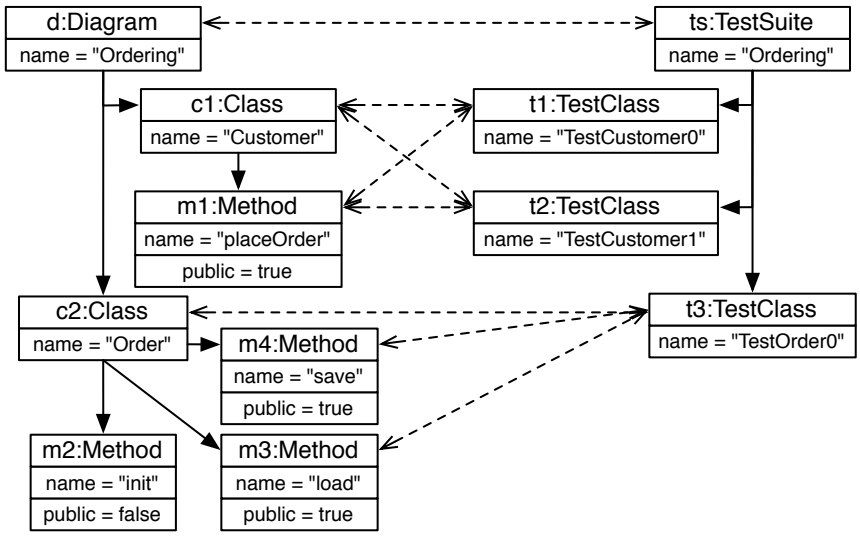


Figure 2: Example consistent triple

This is a model and not a pattern, so small letters are used for node names, and attribute values are depicted with a single equals sign, e.g., `name = Ordering`. Correspondence links are depicted as dashed bidirectional arrows with the assumption that they are of the correct type.

An informal consistency specification can be formalised with a set of TGG rules that together generate the language of all consistent triples such as the triple depicted in Fig. 2. Two simple TGG rules `R1` and `R2` for the running example are depicted in Fig. 3. A TGG rule consists of two patterns $L \rightarrow R$, where $L$ is the precondition and $R$, an extension of $L$, is the postcondition for applying the rule. Elements in $L$ are called *context* elements as they must be present to apply the rule, while elements in $R \setminus L$ are called *created* elements as they are created as a result of applying the rule. To represent rules, a compact concrete syntax is used, merging both patterns $L$ and $R$ into a single pattern by using colours and markup to distinguish elements: context elements are black, while created elements are green with an additional "++" markup. As with the types of links, this markup is only depicted when absolutely necessary, e.g., all links connected to a created object must be also created and do not have any explicit markup.

The rule `R1` (to the left of Fig. 3) creates a consistent pair of a diagram and a test suite, demanding that they have the same name via the simple attribute condition `D.name == TS.name`. Rules such as `R1` are referred to as *island rules* [ALK+15] as they do not require context and are thus always applicable, resulting in an "island" that is not connected to anything else. The rule `R2` (to the right of Fig. 3) demands a consistent pair of a diagram and a test suite, and extends this pair by creating a class and a test class. A simple attribute condition is used to demand that the name of the test class be formed by adding the suffix "0" to the name of the class. Such rules are called *extension rules* [ALK+15] as they create elements that are directly attached to existing context elements. With just these two TGG rules, a substantial part of the example triple in Fig. 2 can already be created: `R1` can be used to create $d \leftrightarrow ts$, `R2` can be applied two times to create $c1 \leftrightarrow t1$, and $c2 \leftrightarrow t3$. All other elements require advanced TGG language features, which will be discussed in the following sections.

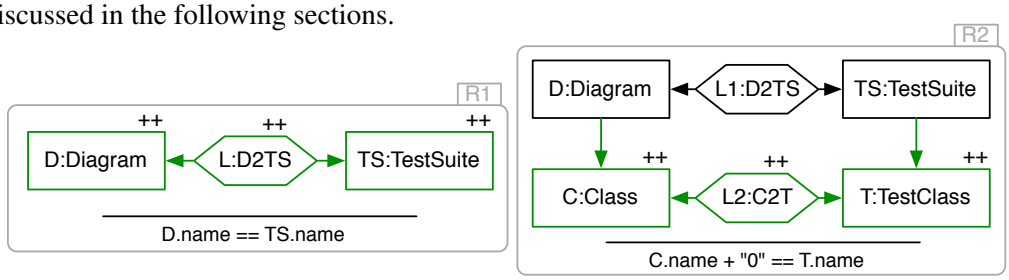


Figure 3: Basic TGG rules for running example

## 3 Research dimensions

Research on TGGs can be viewed in a hyperspace spanned by five dimensions each representing the progress in a certain direction, going from basic to advanced. Every "point" in this hyperspace is thus a specific combination of five properties taken from these dimensions, and can be represented as a trace through a radar chart as depicted in Fig. 4. The points depicted in Fig. 4 represent the state of TGG research in 1995 (as presented in the first paper on TGGs [Sch94]), the current state as of 2015, and our vision for the future.

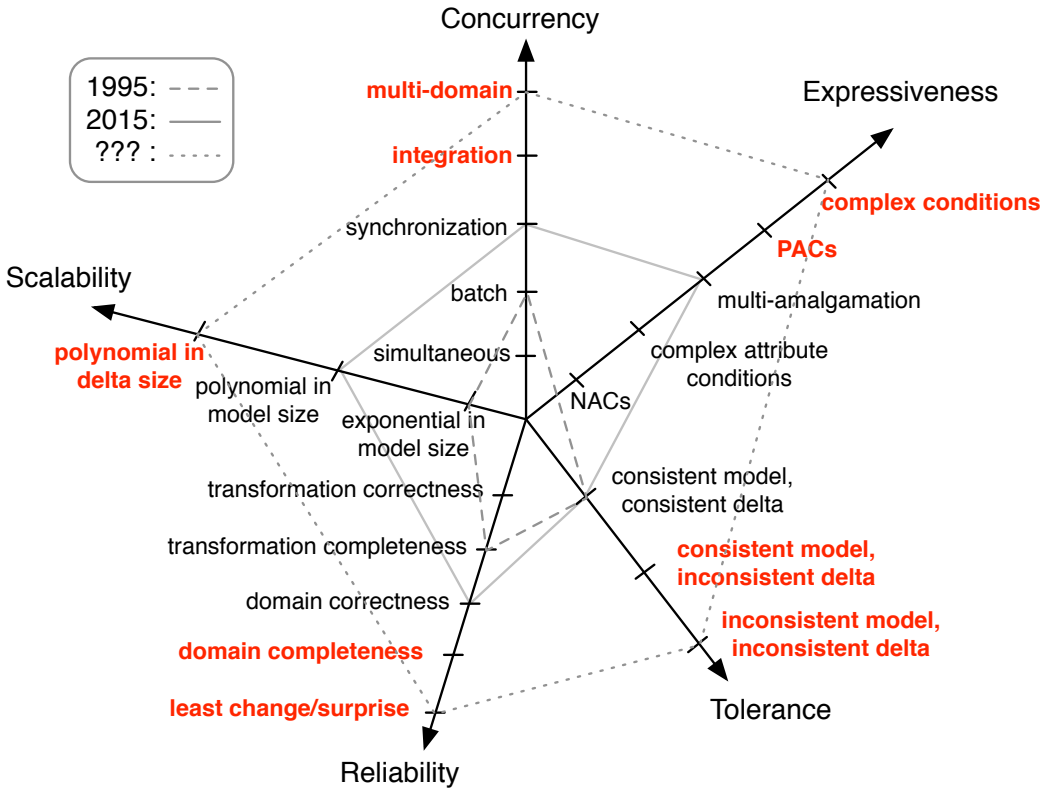


Figure 4: Overview of TGG research dimensions

A brief overview of all research dimensions is provided in the following, before the five dimensions are discussed in detail in corresponding sections 3.1 – 3.5.

*Expressiveness*, defined in this paper as the class of consistency relations that can be expressed with a set of TGG language features, is a crucial dimension as increasing expressiveness is important for the practical applicability of TGGs, i.e., capturing real-world consistency relations as precisely as possible. The expressiveness dimension in Fig. 4 lists TGG language features that have been discussed in existing literature. As we shall see in Section 3.1, our relatively simple running example already requires Negative Application Conditions (NACs), complex attribute conditions, and even multi-amalgamation.

The *concurrency* dimension represents the various operational scenarios in which TGGs can be used. The main idea is to induce a high-level consistency relation with the TGG as follows: *A pair of source and target models are consistent, if they can be generated as part of a triple using the rules of the TGG.* This specification is then adjusted appropriately (operationalised) so it can be used to accomplish different tasks. Typical operational scenarios are discussed in Section 3.2.

To ensure the practical applicability of TGGs, *scalability* is important so that real-world models of realistic size can be handled appropriately. Scalability with respect to relevant factors (model size, size of changes to be propagated) is discussed in Section 3.3.

The *reliability* dimension comprises desirable formal properties that can be guaranteed for the different TGG-based operational scenarios. In Section 3.4, we discuss the most established properties in this dimension, and give references to different existing formalisations.

The focus up until now in TGG research has been primarily on building fully automated transformations that guarantee consistent output for consistent input, i.e., all formal properties (cf. reliability dimension) are typically only guaranteed for consistent input. As argued by Stevens in [Ste14], however, models might never be fully consistent in practice, with engineers working concurrently and using tool support to improve (but not necessarily enforce!) consistency at regular steps. Related challenges for TGG research are discussed in Section 3.5.

## 3.1 Expressiveness

**Negative Application Conditions (NACs):** Consider our running example and the TGG consisting of the schema depicted in Fig. 1 and TGG rules `R1` and `R2` (Fig. 3). Apart from not yet being able to fully create the example triple depicted in Fig. 2, `R2` can be applied multiple times to create classes with the same name. This violates the negative source domain constraint `nc` (Fig. 1) and should be prevented. NACs are a well-known feature from graph transformation used to control rule applicability, and have been studied extensively in the context of TGGs [EHS09, GEH11, HEGO10, KLKS10, AST12]. `R2` can be extended with a NAC that forbids the presence of a class with the same name as the class to be created. Such an extension is depicted in Fig. 5 as `R2'`. Negative elements are depicted as "crossed out" and extend the precondition of the rule, blocking rule application if they can be found in a model.

Transferring NACs to TGGs is, however, not as straightforward as it might seem. In contrast to normal graph transformation, the set of TGG rules is typically used as a high-level specification of consistency to derive *operational* transformations (cf. the concurrency dimension) and it is challenging to extend the various operationalisations to take NACs into account. Most approaches restrict the usage of NACs in some way or the other, e.g., to source and target NACs consisting of only source or target elements, respectively.

Depending on the TGG approach, NACs can also have an adverse effect on scalability and, e.g., [AST12] restricts the usage of NACs to only enforcing negative domain constraints as in `R2'`. Indeed, some approaches with a primary focus on scalability have explicitly chosen not to provide any support for NACs, e.g., [GHL14]. NACs are, nevertheless, an important and useful TGG language feature and the current support for NACs can still be improved especially regarding lifting current restrictions.

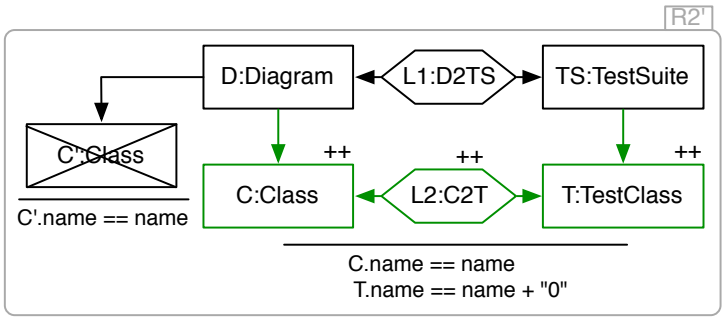


Figure 5: Using a NAC to control rule application

**Complex Attribute Conditions:** In any practical application, specifying the consistency of attribute values cannot be avoided. As algebraic graph transformation is generally more suitable for specifying *structural* rather than attribute-related relationships [Ren10], integrating a flexible handling of complex attribute conditions in TGGs has been a long standing challenge. Some of the difficulties involved are discussed in [KW07, LHGO12].

Allowing arbitrary OCL constraints in TGG rules [KRW04] certainly increases expressiveness, but also makes it harder (perhaps impossible) to guarantee any formal properties (cf. reliability dimension). Another approach is to use the attribute constraints in TGG rules as an "interface", i.e., only to check the conformance of manually implemented operationalisations (e.g., in OCL or plain Java) [GHL14]. A further approach is to integrate black-box constraints in TGG rules, guaranteeing that such atomic constraints can be mixed and combined as required to formulate complex constraints in each rule [AVS12]. This allows for both user-defined, problem-specific constraints (implemented in, e.g., Java) as well as a supplied set of library constraints.

To handle our running example, a constraint `addUniqueIndex(className, testClassName)` can be implemented as a user-defined constraint and used in a TGG rule that creates a new test class for an existing class. This is depicted as rule `R3` to the left of Fig. 6 (ignore `R4` for the moment). Depending on the operationalisation, the constraint implementation might have to deal with (i) deciding if a given class name and test class name are consistent, (ii) deriving a consistent test class name from a given class name by post-pending a unique index, (iii) deriving a consistent class name from a given test class name by simply stripping off the index, and finally (iv) generating consistent pairs of class names and test class names. Depending on how a constraint is combined with other constraints, however, note that not all possible cases must be supported. The "feasibility" of a combination of constraints can be checked at runtime when operationalising the rules [AVS12].

Support for complex attribute conditions in TGG rules is now supported in most TGG tools to a certain extent, but again this can still be improved, especially regarding corresponding formal analysis techniques (cf. reliability dimension), which often do not take attribute values and conditions into account (cf. [DV14] for a discussion and ongoing work on this).

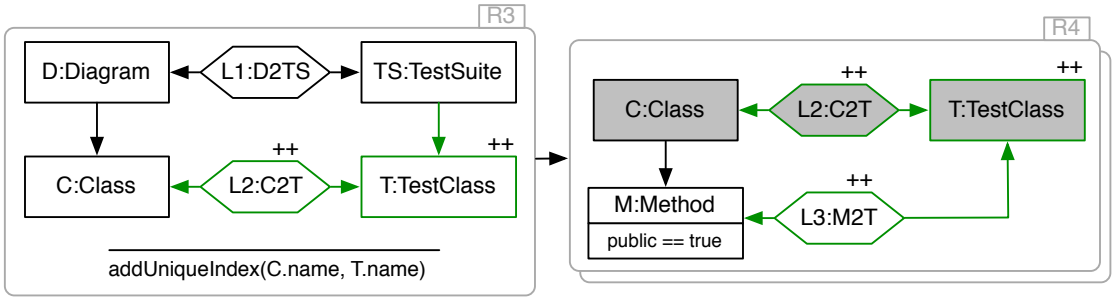


Figure 6: Rules for adding additional test classes

**Multi-Amalgamation:** Considering our running example and the triple depicted in Fig. 2, we can now create all test classes with appropriate attribute values using `R3`. Whenever an additional test case is created, however, a correspondence link is supposed to be created *to all*

*public methods* of the class. As `c1` only has a single method `m1`, creating the correspondence link $m1 \leftrightarrow t2$ could be accomplished by extending `R3` and adding an additional rule for this case. In the case of $c2$ that has two public methods, however, two links $m3 \leftrightarrow t3$ and $m4 \leftrightarrow t3$ have to be created meaning that a further extension of `R3` is again required. This will not work in general as classes can have arbitrarily many public methods.

A solution to the problem of establishing *1-to-m* relations in a single rule application is provided by multi-amalgamation [BFH87, GEH10], a well-known language feature from graph transformation. Progress on transferring multi-amalgamation to TGGs has been made [LAST15, LAS15] and the so-called *multi-rule* `R4` depicted in Fig. 6 can be used to handle the creation of correspondence links to existing public methods. The main idea is to generalise TGG rules to *interaction schemes*, which consist of a *kernel* rule that is applied once, and a number of multi-rules (denoted in the concrete syntax with a double border and an incoming arrow from the kernel rule) that are applied as often as possible in such a manner that all applications agree with the kernel application. In this case, the kernel is `R3`, which is applied to create a new test class for an existing class, while `R4` is the multi-rule that is applied as often as possible (creating a correspondence link to every existing public method of the class). Multi-rules contain and extend the entire kernel, but only necessary elements, emphasised with a grey background, need to be specified in the concrete syntax.

To complete our running example, we still have to add rules that create methods and connect them to their classes. Interestingly, this also requires multi-amalgamation as a newly added public method must be connected to all existing test classes. Figure 7 depicts an interaction scheme consisting of kernel `R5` to add a new public method, and multi-rule `R6` to create correspondence links to all existing test classes. Finally, a so-called *ignore rule* `R7` [ALK+15] is used to specify that adding private methods does not affect the target or correspondence models in any way.

Support for multi-amalgamation is still work in progress and requires further research, especially regarding the development of the fundamental theory and related static analyses [TG15].

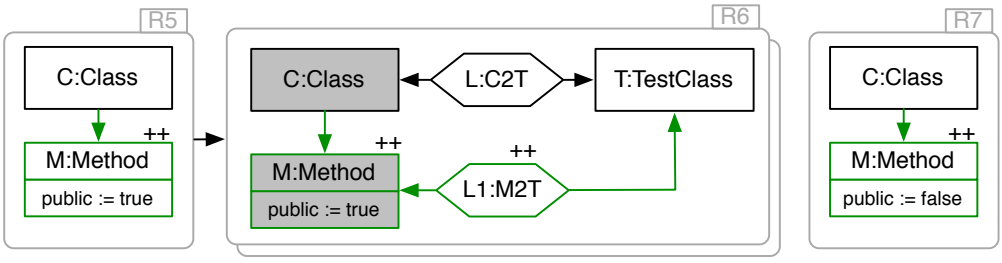


Figure 7: Rules for adding methods

**Positive and Complex Application Conditions:** The final two points on the expressiveness axis can be viewed as challenges for future TGG research. Positive Application Conditions (PACs) generalise NACs and are required in TGG rules, e.g., to ensure that positive constraints such as `pc` (Fig. 1) are not violated. As NACs can only ensure upper bounds of multiplicities, PACs are also required to ensure lower bounds. PACs can be further generalised to complex application conditions, providing support for *nesting* of conditions. This has been shown to be

equivalent to first order logic and covers a useful subset of OCL [RAB+15]. Although complex application conditions have been integrated into TGGs in [GEH11], this is only a first step in the direction as the operationalisation of such TGG rules and corresponding algorithms for the various application scenarios (cf. concurrency dimension) are not discussed.

## 3.2 Concurrency

**Simultaneous Creation of Consistent Triples:** This might seem surprising but applying TGG rules directly as specified can already be useful. This "simultaneous" creation of triples can, for example, be used to implement a syntax directed visual modelling tool with separate models for concrete syntax and semantic interpretation [LGB07]. In this case, the TGG models the synchronised evolution of both models in parallel.

Another application for the simultaneous creation of triples is for test generation [WAS14, HLG+11]. The idea here is to view the TGG as a high-level specification of correctness, i.e., a "test model" for a transformation that can be implemented manually in any language of choice. Even though such a transformation could be automatically derived from the TGG, in some cases it still makes sense to do this manually: (i) the TGG tool in use does not support the target platform, (ii) extra possibly problem-specific optimisations are necessary for efficiency reasons, or (iii) the TGG is incomplete in the sense that only certain (very important) aspects of the transformation are modelled and are to be tested.

Supporting the simultaneous creation of triples is straightforward but, as is often the case, the devil is in the details. Challenges include supporting NACs efficiently, implementing complex attribute conditions as high-quality attribute value generators, and guiding the generation process to produce "balanced" or realistic models with respect to the concrete task or domain. Some of these challenges are discussed in [WAS14].

**Batch Transformations:** A considerable amount of work has been invested in supporting batch forward and backward transformations, which are derived automatically from a TGG. The main idea is to derive a forward transformation `FWD` that is able to extend an existing source model to a complete triple, i.e., appending a consistent correspondence and target model. This is depicted for our running example in Fig. 8 and applies analogously to a backward transformation `BWD`. These operational scenarios are referred to as "batch" transformations as the output is created from scratch, and the entire input is processed afresh, independent of former results.

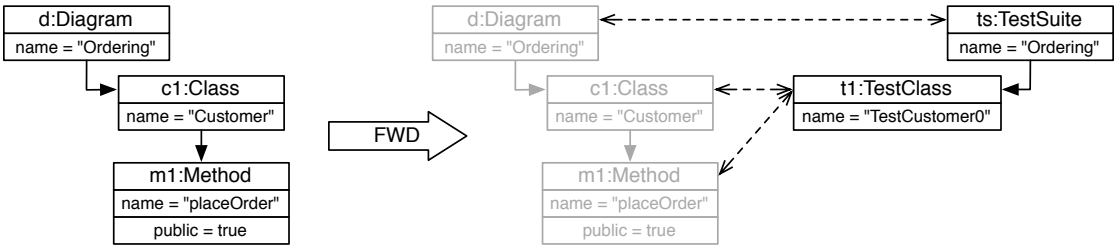


Figure 8: A batch forward transformation

There exist theoretical results characterising precisely when a TGG can be operationalised to yield forward and backward transformations, and when given forward and backward transformations conform to a TGG [EEE+07]. This is particularly important when considering or introducing new language features as it might no longer be possible to derive a forward/backward transformation that retains the semantics of the TGG, i.e., conforms to the TGG.

Different approaches can be used to derive forward and backward transformations, ranging from fully interpretative approaches [KRW04], to "compiling" the TGG to yield a set of "forward" graph transformation rules that can be directly executed to realise the forward transformation [EH12]. Most approaches take a mixed approach, deriving forward/backward rules and controlling their application with an underlying TGG control algorithm [GHL14, LAS14a]. A comparison of TGG approaches with a focus on batch transformations is provided in [HLG+13].

Batch transformations are certainly relevant and useful in practise [HGN+13], but it is difficult to argue why TGGs are better than a straightforward combination of unidirectional model transformation languages. Arguments do include increased productivity and consistency by construction, but these can be guaranteed for a combination of separate unidirectional approaches, effectively "faking" bidirectionality [PDPR14] without a *bx* tool. Indeed, batch forward and backward transformation might not be a very convincing primary application scenario for TGGs, especially as unidirectional approaches are often more efficient [LAS14a].

**Model Synchronisation:** The true potential of TGGs lies more in realising incrementally working *synchronisation* tools, already mentioned as future work in the first paper on TGGs [Sch94]. A *forward model synchroniser* $Sync_F$ takes a triple and a change to the source model $\Delta_S$ (referred to as a *source delta*) as input, and produces a consistent output triple by manipulating the existing correspondence and target models. This means that $\Delta_S$ is propagated to yield corresponding $\Delta_C$ and $\Delta_T$ in a manner that conforms to the underlying TGG. This is depicted schematically in Fig. 9 for our running example (analogously for backward model synchronisers). Note that $\Delta_S$ consists of an attribute change (`c1.name` is changed from `Client` to `Customer`), an addition of an object (method `m1`), and a link (connecting `c1` to `m1`).

The challenge here is to work *incrementally*, taking the entire input triple into account and only changing what is necessary. This is indicated in Fig. 9 by greying out unaffected parts. Model synchronisation is a generalisation of the batch case as batch transformations correspond to the empty triple as input, and creating an entire source or target model as the source/target delta to be propagated. Model synchronisation cannot be "faked" so easily by combining unidirectional transformations, as it is impossible to avoid information loss in general without accessing the old state of the models involved. In our example, applying a batch forward transformation to the source model in Fig. 9 would result in the same triple depicted in Fig. 8, i.e., test class `t2` would not be created. The reason here is that it is impossible to know how many additional test classes to create for each class in the source model, solely by inspecting the source model.

Building well-behaved incremental synchronisation tools is challenging, and establishing a theoretical foundation for this task is a current focus of the *bx* community [CFH+09, Ste08]. Practical scenarios that require synchronisation are often "symmetric" in the sense that both domains contain data that is irrelevant for consistency but must not be discarded in the synchronisation process [DWGC14]. TGG-based formal synchronisation frameworks (e.g., [HEO+11]),

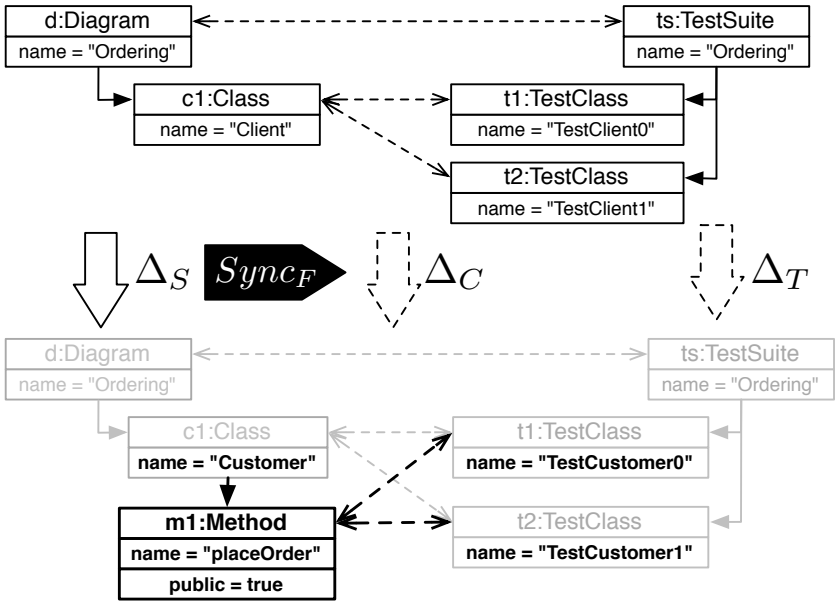


Figure 9: A forward synchronisation

as well as multiple implementations (cf. [LAS+14b] for a comparison) exist. The different approaches apply various strategies and pose certain restrictions to make a compromise between efficiency, being as incremental as possible, and guaranteeing well-behavedness. Improving the existing support for model synchronisation is a current focus of TGG research.

**Model Integration:** Model synchronisation can be further generalised to *model integration*, where the task is not just to propagate a source or target delta, but to consolidate a given pair of source *and* target deltas applied independently to the same triple of models. Model integration is required to support concurrent engineering activities, where engineers work for a "sprint" in parallel on multiple models and then restore consistency at given points in time (cf. [Anj14, RLSS11]). Model integration thus requires some form of conflict detection and resolution as the pair of source and target deltas cannot be expected to be consistent in any sense.

Existing frameworks for model integration all follow the straightforward idea presented by [XSHT13] of combining forward and backward model synchronisers with a model merge: source and target deltas are propagated to the respective other side so they can be merged with the changes applied directly there. The TGG-based model integration framework of [HEEO12] combines this idea with existing theory of conflict detection and resolution provided by [EET11]. These "propagation based" frameworks for model integration are far from ideal (see [OBE+13] for a comparison and discussion) and share the common problem of treating the forward and backward synchronisation steps in a black-box manner, i.e., not being to control them for optimal conflict avoidance. A more "bidirectional" approach is suggested by [OBE+13] as a possible improvement. The TGG-based frameworks are also all currently only theoretical and are yet to be implemented and practically evaluated with respect to, e.g., scalability and expressiveness.

Model integration can be seen as *the* application scenario for *bx* in general and TGGs in particular. It is relatively easy to argue the added value of a *bx* language with direct support for model integration, as opposed to somehow combining unidirectional transformation languages. Model integration is, therefore, a promising current and future focus of TGG research.

**Multi-Domain TGGs:** In practice, even though some cases might involve only two related models, supporting concurrent engineering activities in general will involve maintaining the consistency of a whole *set* of related models. The *bx* community is currently only considering the simplified case of handling a pair of related models, not because this is all that is required, but simply because this simplifies the problem, which is already complex enough.

Nonetheless, [KS06, TA15] have already shown that at least the basic theory for TGGs can be extended elegantly from triples to arbitrarily connected networks of models. All TGG tools we are aware of, however, still consider multi-domain TGGs to be currently out-of-scope.

### 3.3 Scalability

An advantage of TGGs compared to, e.g., constraint solver-based solutions is the potential to scale better. This is due to the considerable experience in the graph transformation community with developing scalable graph pattern matching engines, which have been successfully leveraged for TGGs [LAS+14b, HLG+13].

The underlying problem of "graph parsing", however, remains difficult and naïve solutions can easily explode exponentially, even for TGGs with only a few rules, and moderate model sizes. In practice, therefore, the class of supported TGGs is typically restricted in some suitable way to guarantee scalability [ALST14, HEGO10, KLKS10].

There are two relevant factors when discussing scalability of TGG-based tools: (i) the average size (number of elements) of (rule) patterns $k$, and (ii) the size (number of elements) of involved models $n$. To be useful in practice, current TGG tools make a tradeoff between expressiveness (e.g., restricting the usage of NACs) and scalability, ensuring at least polynomial runtime in model size, i.e., at most $O(n^k)$.

The challenge for future work is to completely decouple synchronisation/integration time from model size, i.e., to scale polynomially with respect to the number of elements changed $n'$, where $n'$ is normally much smaller than $n$. Approaches such as [JKS06, ARDS14, GHL14] have shown that this can be attained if the set of supported TGGs is severely restricted. Further relaxing these restrictions and still scaling with respect to delta size (perhaps at least in most "important" cases) is ongoing and future work.

### 3.4 Reliability

To simplify the following discussion, models will denoted by $G_X$ ($G$ for *graph*), where $X \in \{S,C,T\}$ represents the domain (source, correspondence, or target) of the model. Triples of models are consequently denoted by $G_S \leftarrow G_C \rightarrow G_T$. The rules of a TGG generate a language of consistent triples, denoted by $\mathscr{L}(TGG)$. With this notation, we can now formulate TGG-based consistency succinctly as: $G_S, G_T$ are consistent $\Leftrightarrow \exists\, G_S \leftarrow G_C \rightarrow G_T \in \mathscr{L}(TGG)$.

**Transformation Correctness and Completeness:** Let us refer to one of the operationalisations on the concurrency axis as a *transformation* $\tau$. A fundamental requirement is that such a derived transformation $\tau$ conform to its underlying TGG:

A transformation $\tau$ is *transformation correct* if it produces consistent triples only.

This is formalised differently depending on the exact operationalisation: for batch transformations cf. [KLKS10, SK08, EEE+07], for model synchronisation [HEO+11, LAVS12], and for model integration [HEEO12, OBE+13]. Considering our running example, transformation correctness guarantees that the result of executing a synchronisation (Fig. 9) will always be a triple that could also be created from scratch by applying the rules of the TGG (Fig. 3, 6, 7).

A related and equally desired property of transformations is *totality*, i.e., demanding that the domain of a transformation $\tau$ be precisely stated and that $\tau$ be total on its set of valid inputs:

$\tau$ is *transformation complete* if it is total on a precisely defined set of valid inputs.

Although not demanded explicitly, "precisely defined" often means a set of conditions that can be checked at design time via static analysis techniques. The practical relevance of transformation completeness is that when an implementation throws an exception, this can only mean that the input was invalid (and this can possibly be ruled out with a corresponding static analysis).

Once again, transformation completeness is formalised differently depending on the operational scenario: for batch transformations cf. [KLKS10, SK08, EEE+07], and for model synchronisation [HEO+11, LAVS12]. For our running example, transformation completeness guarantees that *any* source delta that leads to a source model, which is part of a consistent triple, can be forward synchronised (Fig. 9). Note that the concept of transformation completeness does not seem to make sense in the more general context of model integration, where a host of properties have been suggested to formulate reasonable expectations on the conflict detection and resolution process instead [HEEO12, OBE+13]. Indeed, capturing an analogous completeness property for model integration can be viewed as work in progress.

**Domain Correctness and Completeness:** When working with models and metamodels, the rules of a TGG are not the sole source of consistency. The domains of the source and target models also comprise "domain" constraints, which should not be violated by valid or *domain consistent* models. According to [EEPT06], a triple of metamodels can be formalised as a *type triple graph* $TG$ together with a set $\mathscr{C}$ of domain constraints. In analogy to $\mathscr{L}(TGG)$, $\mathscr{L}(TG)$ denotes the set of all well-typed triples, while $\mathscr{L}(TG,\mathscr{C})$ denotes the set of all *domain consistent* triples that are well-typed *and* additionally do not violate any domain constraints from a given set of constraints $\mathscr{C}$.

Given these independent sources of consistency requirements, $\mathscr{L}(TGG)$ and $\mathscr{L}(TG,\mathscr{C})$, the following two properties are used to demand the "compatibility" of a TGG with its domains:

A TGG is *domain correct* if: $G \in \mathscr{L}(TGG) \Rightarrow G \in \mathscr{L}(TG,\mathscr{C})$.

This means that transformation correctness is sufficient for domain consistency. *Domain completeness* requires that transformation correctness also be *necessary* for domain consistency in *both* the source and target domains. To express this for the source domain (applies analogously

to the target domain), let $\mathscr{L}(TG_S, \mathscr{C}_S)$ denote the set of all *source domain* consistent models, i.e., well-typed and valid source models according to the source type graph $TG_S$ and given set of source domain constraints $\mathscr{C}_S$:

A TGG is *source domain complete* if: $G_S \in \mathscr{L}(TG_S, \mathscr{C}_S) \Rightarrow \exists\, G_S \leftarrow G_C \rightarrow G_T \in \mathscr{L}(TGG)$.

A static analysis technique for ensuring domain correctness given a TGG and a set of domain constraints is presented in [AST12], while initial results on analysing domain completeness is given by [EHSB13]. In both cases, an extension to advanced language features (cf. expressiveness dimension), an implementation, and integration in a TGG tool is still work in progress. The TGG for our running example is *not* domain correct, as it can create classes without any public methods using `R2'` (Fig. 5). The analysis of [AST12] would identify this violation, and even suggest extending `R2'` to create a public method together with every new class C. The TGG is also not domain complete, as numerous target domain constraints are missing, e.g., to ensure that the names of test classes are unique and can actually be generated by the TGG.

The practical relevance of domain correctness is to ensure that TGG-based tools take domain constraints adequately into account. A check for domain completeness can be used to indicate that certain domain consistent models cannot yet be generated by the TGG. This is often unwanted as it points to implicit domain constraints. In certain operational scenarios, domain completeness can also be exploited as a means of checking for input validity: for a batch forward transformation that is both transformation complete and domain complete, a source model is valid input if it is source domain consistent.

**Least Change, Least Surprise:** Especially relevant in the general case of model integration, the formal property of least change or least surprise is meant to capture the perceived *quality* of the synchronisation / integration process. Transformation correctness is only a very basic requirement and does not pose any restrictions on *how* consistency is restored. For instance, based solely on transformation correctness, i.e., ignoring runtime, there are TGGs for which it is impossible to distinguish a forward synchroniser that simply deletes all correspondence / target elements and always performs a batch forward transformation, from a forward synchroniser that works incrementally and only performs minimal changes. Compare the case of our running example depicted in Fig. 8 and Fig. 9: Retaining test class $t2$ during synchronisation is obviously desirable (Fig. 9), yet there is nothing stating formally that this is better than the result in Fig. 8!

The intuition here is to prefer synchronisers that restore consistency by "changing as little as possible". This turns out, however, to be challenging to formalise and is a current focus of *bx* research. Amongst other factors, the "best behaviour" of a synchroniser is not necessarily directly connected to the "size" of the changes applied according to, e.g., some form of metric, but might be more related to being as predictable as possible, i.e., causing least *surprise* as opposed to afflicting least change. The interested reader is referred to [CGMS15] for a detailed overview of related work in this context as well as a discussion of challenges.

In some (simple) cases, the *round-tripping laws* of the delta lens framework [Dis08], often capture desirable behaviour better than transformation correctness. First steps at comparing the lens laws with TGGs have been taken by [HEO+11], but only for the simplified case of TGGs with *functional behaviour*, i.e., rule `R3` in our running example would not be allowed as it can be applied arbitrarily often, leading to the discrepancy between batch (Fig. 8) and sync (Fig. 9).

### 3.5 Tolerance

The current trend, especially for model integration, is towards tools that *tolerate* or better yet embrace inconsistencies [Ste14]. As indicated in the tolerance dimension (Fig. 4), a first step in this direction could involve accepting inconsistent deltas but ensuring that the resulting triple is always consistent. In many cases, however, this might not be what is wanted at all. It might actually be better for a tool to only *improve* the situation if possible, and not enforce full consistency (which could mean deleting everything in an extreme case) [Ste14].

For *bx* tools to be of any use in practice, a fine grained notion of consistency should be supported, i.e., not consistent or inconsistent, but providing something akin to a "lattice" of consistency, together with formal guarantees to ensure that the *bx* tool either improves the situation with respect to consistency, or does nothing. The argument is that in practice, models might never be fully consistent, with engineers working constantly and concurrently on them [Ste14, CGMS15].

This is a completely new challenge for TGG research, as all formal results are currently based on a fully consistent starting point. It is still unclear how to provide a fine grained notion of consistency for TGGs and how best to integrate a concept of "tolerance" in the current formal framework. In combination with model integration, providing support for tolerance is perhaps *the* primary future challenge that TGG research must address.

## 4 Related work

The goals and scope of the *bx* community are outlined in [CFH+09]. A survey of *bx* approaches is provided by [Ste08], while a more recent feature-based *bx* taxonomy is given by [HTCH15]. A complementary classification of *bx* scenarios is presented in [DWGC14]. While all these papers present and classify research in *bx*, giving a vision in some cases, e.g., support for increasingly symmetric *bx* scenarios in [DWGC14], our paper focusses solely on TGG research, and is thus able to discuss TGG-specific issues and challenges in more detail.

A proposal and vision for a repository of *bx* examples is presented in [CMSG14]. The *bx* repository has since then been established and is now a source for a growing number of *bx* examples, including numerous TGG specifications. While [CMSG14] does not explicitly discuss TGGs, the repository serves as a source for examples that cannot or can only partially be specified using TGGs, motivating new language features, application scenarios, and formal properties. Visionary papers in the context of *bx* such as [Ste14] on tolerance, and [CGMS15] on the principle of least change / surprise, have had a substantial influence on TGG research. In this paper, we have discussed these challenges for TGGs, stating relevant current and future work.

There are currently two TGG tool comparison papers: [HLG+13] for general aspects, and [LAS+14b] to focus primarily on support for model synchronisation. This paper complements these practical and often technical, tool-specific comparisons, by providing a more general discussion and a vision for future TGG research.

This paper can be seen as a continuation of [Sch94], already with a vision for "incrementally working translators", and [SK08], summarising 15 years of TGGs and focussing on challenges concerning mainly TGG-based batch transformations. In contrast to [SK08], we have chosen to distinguish two different kinds of correctness/completeness: transformation correctness/completeness and domain correctness/completeness. We have also posed, for the first time

for TGGs, challenges concerning tolerance and least change/surprise, both currently areas of active research in the *bx* community. Finally, we have identified model integration, considered clearly out-of-scope in [SK08, Sch94], as a primary focus for future TGG research.

## 5 Summary and conclusion

In this paper, we have given an overview of the current state-of-the-art in TGG research, organising the different directions in five research dimensions with past, current, and future challenges. As a conclusion, we believe that the primary potential of TGGs in the years to come lies in realising tolerant, scalable, and reliable tools for *model integration*.

Some practical but equally important points we did not have space to discuss include: extending the current support for the *modularity* of TGG specifications [GR12, KKS07, ASLS14], establishing *debugging* frameworks, providing ample *documentation* (handbooks, examples, tutorials), and improving *tool support* in general. Some of these issues are discussed in our TGG tool comparison papers [LAS+14b, HLG+13].

## Bibliography

[ALK+15] A. Anjorin, E. Leblebici, R. Kluge, A. Schürr, P. Stevens. A Systematic Approach and Guidelines to Developing a Triple Graph Grammar. In Cunha and Kindler (eds.), *BX 2015*. CEUR Workshop Proceedings 1396, pp. 81–95. CEUR-WS.org, 2015.

[ALST14] A. Anjorin, E. Leblebici, A. Schürr, G. Taentzer. A Static Analysis of Non-Confluent Triple Graph Grammars for Efficient Model Transformation. In Giese and König (eds.), *ICGT 2014*. LNCS 8571, pp. 130–145. Springer, 2014.

[Anj14] A. Anjorin. *Synchronization of Models on Different Abstraction Levels using Triple Graph Grammars*. Phd thesis, Technische Universität Darmstadt, 2014.

[ARDS14] A. Anjorin, S. Rose, F. Deckwerth, A. Schürr. Efficient Model Synchronization with View Triple Graph Grammars. In Cabot and Rubin (eds.), *ECMFA 2014*. LNCS 8569, pp. 1–17. Springer, 2014.

[ASLS14] A. Anjorin, K. Saller, M. Lochau, A. Schürr. Modularizing Triple Graph Grammars Using Rule Refinement. In Gnesi and Rensink (eds.), *FASE 2014*. LNCS 8411, pp. 340–354. Springer, 2014.

[AST12] A. Anjorin, A. Schürr, G. Taentzer. Construction of Integrity Preserving Triple Graph Grammars. In Ehrig et al. (eds.), *ICGT 2012*. LNCS 7562, pp. 356–370. Springer, 2012.

[AVS12] A. Anjorin, G. Varró, A. Schürr. Complex Attribute Manipulation in TGGs with Constraint-Based Programming Techniques. In Hermann and Voigtländer (eds.), *BX 2012*. ECEASST 49, pp. 1–16. EASST, 2012.

[BFH87] P. Boehm, H.-R. Fonio, A. Habel. Amalgamation of graph transformations: A synchronization mechanism. *JCSS* 34(2-3):377–408, 1987.

[CFH+09] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, J. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In Paige (ed.), *ICMT 2009*. LNCS 5563, pp. 260–283. Springer, 2009.

[CGMS15] J. Cheney, J. Gibbons, J. McKinna, P. Stevens. Towards a Principle of Least Surprise for Bidirectional Transformations. In Cunha and Kindler (eds.), *BX 2015*. CEUR Workshop Proceedings 1396, pp. 66–80. CEUR-WS.org, 2015.

[CMSG14] J. Cheney, J. McKinna, P. Stevens, J. Gibbons. Towards a Repository of Bx Examples. In Candan et al. (eds.), *Proceedings of the Workshops of EDBT/ICDT 2014*. CEUR Workshop Proceedings 1133, pp. 87–91. CEUR-WS.org, 2014.

[Dis08] Z. Diskin. Algebraic Models for Bidirectional Model Synchronization. In Czarnecki et al. (eds.), *MoDELS 2008*. LNCS 5301, pp. 21–36. Springer, 2008.

[DV14] F. Deckwerth, G. Varró. Attribute Handling for Generating Preconditions from Graph Constraints. In Giese and König (eds.), *ICGT 2014*. LNCS 8571, pp. 81–96. Springer, jul 2014.

[DWGC14] Z. Diskin, A. Wider, H. Gholizadeh, K. Czarnecki. Towards a Rational Taxonomy for Increasingly Symmetric Model Synchronization. In Ruscio and Varró (eds.), *ICMT 2014*. LNCS 8568, pp. 57–73. Springer, 2014.

[EEE+07] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, G. Taentzer. Information Preserving Bidirectional Model Transformations. In Dwyer and Lopes (eds.), *FASE 2007*. LNCS 4422, pp. 72–86. Springer, 2007.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[EET11] H. Ehrig, C. Ermel, G. Taentzer. A Formal Resolution Strategy for Operation-Based Conflicts in Model Versioning Using Graph Modifications. In Giannakopoulou and Orejas (eds.), *FASE 2011*. LNCS 6603, pp. 202–216. Springer, 2011.

[EH12] C. Ermel, F. Hermann. Visual Modeling and Analysis of EMF Model Transformations Based on Triple Graph Grammars. *ECEASST* 54:1–12, 2012.

[EHS09] H. Ehrig, F. Hermann, C. Sartorius. Completeness and Correctness of Model Transformations Based on Triple Graph Grammars with Negative Application Conditions. In Boronat and Heckel (eds.), *GT-VMT 2009*. ECEASST 18. EASST, 2009.

[EHSB13] H. Ehrig, F. Hermann, H. Schölzel, C. Brandt. Propagation of constraints along model transformations using triple graph grammars and borrowed context. *VLC* 24(5):365–388, 2013.

[GEH10] U. Golas, H. Ehrig, A. Habel. Multi-Amalgamation in Adhesive Categories. In Ehrig et al. (eds.), *ICGT 2010*. LNCS 6372, pp. 346–361. Springer, 2010.

[GEH11] U. Golas, H. Ehrig, F. Hermann. Formal Specification of Model Transformations by Triple Graph Grammars with Application Conditions. In Echahed et al. (eds.), *GCM 2010*. ECEASST 39. ASST, 2011.

[GHL14] H. Giese, S. Hildebrandt, L. Lambers. Bridging the Gap Between Formal Semantics and Implementation of Triple Graph Grammars. *SoSyM* 13(1):273–299, 2014.

[GR12] J. Greenyer, J. Rieke. Applying Advanced TGG Concepts for a Complex Transformation of Sequence Diagram Specifications to Timed Game Automata. In Schürr et al. (eds.), *AGTIVE 2011*. LNCS 7233, pp. 222 – 237. Springer, 2012.

[HEEO12] F. Hermann, H. Ehrig, C. Ermel, F. Orejas. Concurrent Model Synchronization with Conflict Resolution Based on Triple Graph Grammars. In Lara and Zisman (eds.), *FASE 12*. LNCS 7212, pp. 178–193. Springer, 2012.

[HEGO10] F. Hermann, H. Ehrig, U. Golas, F. Orejas. Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars. In Bézivin et al. (eds.), *MDI 2010*. MDI 2010 1866277, pp. 22–31. ACM Press, 2010.

[HEO+11] F. Hermann, H. Ehrig, F. Orejas, K. Czarnecki, Z. Diskin, Y. Xiong. Correctness of Model Synchronization Based on Triple Graph Grammars. In Whittle et al. (eds.), *MoDELS 2011*. LNCS 6981, pp. 668–682. Springer, 2011.

[HGN+13] F. Hermann, S. Gottmann, N. Nachtigall, B. Braatz, G. Morelli, A. Pierre, T. Engel. On an Automated Translation of Satellite Procedures Using Triple Graph Grammars. In Duddy and Kappel (eds.), *ICMT 2013*. LNCS 7909, pp. 50–51. Springer, 2013.

[HLG+11] S. Hildebrandt, L. Lambers, H. Giese, D. Petrick, I. Richter. Automatic Conformance Testing of Optimized Triple Graph Grammar Implementations. In Schürr et al. (eds.), *AGTIVE 2011*. LNCS 7233, pp. 238–253. Springer, 2011.

[HLG+13] S. Hildebrandt, L. Lambers, H. Giese, J. Rieke, J. Greenyer, W. Schäfer, M. Lauder, A. Anjorin, A. Schürr. A Survey of Triple Graph Grammar Tools. In Stevens and Terwilliger (eds.), *BX 2013*. ECEASST 57. EASST, 2013.

[HTCH15] S. Hidaka, M. Tisi, J. Cabot, Z. Hu. Feature-Based Classification of Bidirectional Transformation Approaches. *SoSyM*, pp. 1–22, 2015.

[JKS06] J. Jakob, A. Königs, A. Schürr. Non-Materialized Model View Specification with Triple Graph Grammars. In Corradini et al. (eds.), *ICGT 2006*. LNCS 4178, pp. 321–335. Springer, 2006.

[KKS07] F. Klar, A. Königs, A. Schürr. Model Transformation in the Large. In Crnkovic and Bertolino (eds.), *FSE 2007*. Pp. 285–294. ACM, 2007.

[KLKS10] F. Klar, M. Lauder, A. Königs, A. Schürr. Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In Schürr et al. (eds.), *Festschrift Nagl*. LNCS 5765, pp. 141–174. Springer, 2010.

[KRW04] E. Kindler, V. Rubin, R. Wagner. An Adaptable TGG Interpreter for In-Memory Model Transformations. In Schürr and Zündorf (eds.), *Fujaba Days 2014*. Pp. 35–38. 2004.

[KS06] A. Königs, A. Schürr. MDI: A Rule-based Multi-document and Tool Integration Approach. *SoSym* 5(4):349–368, 2006.

[KW07] E. Kindler, R. Wagner. Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical report, Software Engineering Group, Department of Computer Science, University of Paderborn, 2007.

[LAS14a] E. Leblebici, A. Anjorin, A. Schürr. Developing eMoflon with eMoflon. In Ruscio and Varró (eds.), *ICMT 14*. LNCS 8568, pp. 138–145. Springer, 2014.

[LAS+14b] E. Leblebici, A. Anjorin, A. Schürr, S. Hildebrandt, J. Rieke, J. Greenyer. A Comparison of Incremental Triple Graph Grammar Tools. In Hermann and Sauer (eds.), *GT-VMT 2014*. ECEASST 67. EASST, 2014.

[LAS15] E. Leblebici, A. Anjorin, A. Schürr. Tool Support for Multi-amalgamated Triple Graph Grammars. In Parisi-Presicce and Westfechtel (eds.), *ICGT 2015*. LNCS 9151, pp. 257–265. Springer, 2015.

[LAST15] E. Leblebici, A. Anjorin, A. Schürr, G. Taentzer. Multi-amalgamated Triple Graph Grammars. In Parisi-Presicce and Westfechtel (eds.), *ICGT 2015*. LNCS 9151, pp. 87–103. Springer, 2015.

[LAVS12] M. Lauder, A. Anjorin, G. Varró, A. Schürr. Efficient Model Synchronization with Precedence Triple Graph Grammars. In Ehrig et al. (eds.), *ICGT 2012*. LNCS 7562, pp. 401–415. Springer, 2012.

[LGB07] J. de Lara, E. Guerra, P. Bottoni. Triple Patterns: Compact Specifications for the Generation of Operational Triple Graph Grammar Rules. *ECEASST* 6, 2007.

[LHGO12] L. Lambers, S. Hildebrandt, H. Giese, F. Orejas. Attribute Handling for Bidirectional Model Transformations: The Triple Graph Grammar Case. *ECEASST* 49, 2012.

[OBE+13] F. Orejas, A. Boronat, H. Ehrig, F. Hermann, H. Sch. On Propagation-Based Concurrent Model Synchronization. *ECEASST* 57, 2013.

[PDPR14] C. M. Poskitt, M. Dodds, R. F. Paige, A. Rensink. Towards Rigorously Faking Bidirectional Model Transformations. In Dingel et al. (eds.), *AMT 2014*. CEUR Workshop Proceedings 1277, pp. 70–75. CEUR-WS.org, 2014.

[RAB+15] H. Radke, T. Arendt, J. S. Becker, A. Habel, G. Taentzer. Translating Essential OCL Invariants to Nested Graph Constraints Focusing on Set Operations. In Parisi-Presicce and Westfechtel (eds.), *ICGT 2015*. LNCS 9151, pp. 155–170. Springer, 2015.

[Ren10] A. Rensink. The Edge of Graph Transformation - Graphs for Behavioural Specification. In Engels et al. (eds.), *Festschrift Nagl*. LNCS 5765, pp. 6–32. Springer, 2010.

[RLSS11] S. Rose, M. Lauder, M. Schlereth, A. Schürr. A Multidimensional Approach for Concurrent Model Driven Automation Engineering. In Osis and Asnina (eds.), *Model-Driven Domain Analysis and Software Development: Architectures and Functions*. Pp. 90–113. IGI Publishing, 2011.

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In Mayr et al. (eds.), *WG 1994*. LNCS 903, pp. 151–163. Springer, 1994.

[SK08] A. Schürr, F. Klar. 15 Years of Triple Graph Grammars. In Ehrig et al. (eds.), *ICGT 2008*. LNCS 5214, pp. 411–425. Springer, 2008.

[Ste] P. Stevens. MODELTESTS v0.1 in Bx Examples Repository. http://bx-community.wikidot.com/examples:home.

[Ste08] P. Stevens. A Landscape of Bidirectional Model Transformations. In Lämmel et al. (eds.), *GTTSE 07*. LNCS 5235, pp. 408–424. Springer, 2008.

[Ste14] P. Stevens. Bidirectionally Tolerating Inconsistency: Partial Transformations. In Gnesi and Rensink (eds.), *FASE 2014*. LNCS 8411, pp. 32–46. Springer, 2014.

[TA15] F. Trollmann, S. Albayrak. Extending Model to Model Transformation Results from Triple Graph Grammars to Multiple Models. In Kolovos and Wimmer (eds.), *ICMT 2015*. LNCS 9152, pp. 214–229. Springer, 2015.

[Tae12] G. Taentzer. Instance Generation from Type Graphs with Arbitrary Multiplicities. In Fish and Lambers (eds.), *GT-VMT 2012*. ECEASST 47. EASST, 2012.

[TG15] G. Taentzer, U. Golas. Towards Local Confluence Analysis for Amalgamated Graph Transformation. In *ICGT 2015*. LNCS 9151, pp. 69–86. Springer, 2015.

[WAS14] M. Wieber, A. Anjorin, A. Schürr. On the Usage of TGGs for Automated Model Transformation Testing. In Di Ruscio and Varró (eds.), *ICMT 2014*. LNCS 8568, pp. 1–16. Springer, 2014.

[XSHT13] Y. Xiong, H. Song, Z. Hu, M. Takeichi. Synchronizing concurrent model updates based on bidirectional transformation. *SoSyM* 12(1):89–104, 2013.