Extending Triple Graph Grammars to Formalize Complex View Definitions on Families of Models – Long Version

Lars König
lars.koenig@kit.edu
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

Jens Kosiol
jens.kosiol@b-tu.de
Brandenburgische Technische Universität Cottbus – Senftenberg
Cottbus, Germany

Abstract

View-based development is an important technique to manage complexity and facilitate cooperation in the development of modern and complex systems of systems. This approach places high demands on view definition languages used in the development process. First, such a language needs to be approachable and extremely versatile so that developers from diverse backgrounds are able to use it and can define views for a diverse set of different tasks. Yet, the language also needs to have a precise and formal semantics so that it is possible to integrate the work performed on various views in a controlled manner to obtain a coherent overall system. In this paper, we continue previous work of equipping the NeoJoin view definition language with a formal semantics that is based on triple graph grammars (TGGs). This paves the way for obtaining automated and incremental synchronization procedures between models and views that come with high formal guarantees for their behavior. Simultaneously, our work serves as a further case study of the expressivity and usability of TGGs. We identify one gap, namely convenient support for the translation of overlapping queries from the view definition language, and tackle that gap by introducing a skip semantics for TGG rules, i.e., allowing to skip certain actions a rule prescribes, depending on the application context.

CCS Concepts

• **Software and its engineering → Semantics**; **System modeling languages**; *Traceability*.

Keywords

model views, view definition, triple graph grammars, skip semantics

1 Introduction

Views are an important part of model-driven development to manage the complexity and consistency of models. Both are frequent issues in the development of cyber-physical systems [12]. View definition languages are used to define the available view types, i.e., view metamodels [19], and model-view transformations. A number of view definition languages have been proposed [7], however, these languages lack features desirable for the development of cyber-physical systems, such as expressive transformation operators, and bidirectional and incremental transformations [28].

The NeoJoin language [29] combines the definition of view types and model-view transformations in a query-style, SQL-like language. Its syntax and transformation operators make it accessible for modeling experts as well as software and system engineers. In order to provide a bidirectional and incremental implementation of the transformation operators, König et al. proposed to generate triple graph grammars (TGGs) from the view definitions [27]. TGGs provide a thorough formalism, including forward and backward operationalizations for the grammar's rules that can serve as the basis for automated and incremental synchronization processes. Based on their work, we present a worked out application of TGGs for deriving model views, highlighting prospects and challenges of their application. In detail, we extend the operators defined by König et al. with three special cases: aggregation, cross-product joins, and overlapping queries. We find that TGGs offer high formal guarantees but only limited support for the treatment of attributes and for parsing of the same model element repeatedly to represent it in different ways in the view.

As our second contribution, we extend TGG rules by a *skip semantics*, based on our idea of *effect-oriented graph transformation* [26], and newly develop operationalized variants of TGG rules with skip semantics. We then use the defined skip semantics to suggest TGG rules for overlapping queries. While this paper is confined to continue the foundational translation of queries from the NeoJoin language into TGG rules, we at least briefly highlight which formal results we are confident to obtain based on this in the future.

We start by discussing related work in Section 2 and by introducing our running example and relevant background in Sections 3 and 4. In Section 5, we present more complex queries from the NeoJoin language whose translation into TGG rules remained open in [28], and we develop our solutions to these challenges in Section 6. In Section 7, we conclude, providing a prospect of the formal results that are now in reach, given the translations we develop in this paper. In In Appendices A to C, we provide further technical details and additional examples we omit from the main paper for space reasons.

2 Related Work

Closest related to this paper are (i) other works developing view definition languages and (ii) case studies of TGGs in other contexts. Regarding (i), Bruneliere et al. provide a survey of model view approaches, however, only a limited number of them provides query-style view definitions with bidirectional and incremental model-view transformations [7]. Most notable are OpenFlexo [20]

and VIATRA Viewers [10]. In contrast to NeoJoin [29], neither offers a unified DSL for the definition of view types and model-view transformations, with OpenFlexo providing multiple DSLs and VIATRA Viewers using annotations to define the view type. Closely related to this work is the transformation approach taken by VIATRA Viewers, which builds on EMF-IncQuery [33] and thus uses graph patterns as model queries. The graph patterns are specified in a textual syntax and matched against a model instance using Rete networks [13], which store partial matches and use these to recompute matches upon changes to the model. TGGs have also already been employed to derive views of models [3, 24], however, in these works of Jakob, Anjorin et al., the focus is on restricting the kinds of allowed TGG rules (to gain efficiency) and to not materialize the view (to avoid data duplication) – two topics not further considered in this paper.

Regarding (ii), TGGs have been applied in a range of different scenarios; we here discuss papers that reflect on the pros and cons they experienced when using TGGs. Hermann et al. [23] used TGGs to translate satellite control procedures, Blouin et al. [6] to synchronize between a textual and a visual representation of models in the Architecture Analysis and Design Language; and Buchmann and Westfechtel [8] to bidirectionally and incrementally synchronize between class diagrams and Java code. Generally, these applications of TGGs were successful and their formal underpinning and declarative and approachable nature proved to be advantages. Still, in each of the works the authors had to somehow modify or extend TGGs, in particular to tackle rule management and scalability issues. Finally, Anjorin and Buchmann [1] investigated situations that are especially challenging for TGGs. They identified three situations, namely (i) ignoring certain elements; (ii) the treatment of attributes, in particular, when the same information is represented structurally in one model but numerically in the other; and (iii) related models that are to be parsed in opposite orders. With our introduction of a skip semantics for TGG rules in Section 6.3 we provide a solution for their first challenging situation.

3 Running Example

To illustrate the problems we are facing when implementing our query language with TGGs, we present a small example from the domain of system modeling and simulation. In our example, we have two models: a blocks-and-ports model for simulating hardware components and a model describing the sensors available in our hardware components. We show the metamodel of our blocks-and-ports model in Fig. 1. The top level entity of this *Simulation* metamodel is a *Model*, which contains *Blocks* and *Connections*. As the connections between blocks are not relevant for our examples, we will not explain them in more detail. A *Block*, on the other hand, has a *name* and a number of input and output ports, represented by the classes *InPort* and *OutPort*. Further, a *Block* can be associated to multiple sensors, identified by the attribute *sensorGroup*. Multiple *Blocks* can be associated to the same sensors, i.e., have the same value for the attribute *sensorGroup*. *In-* and *OutPorts* have a *name* and a *type*. In addition, we show our *Hardware* metamodel, describing the available sensors, in Fig. 2. A *Component*, which represents the same thing as the *Model* from our blocks-and-ports

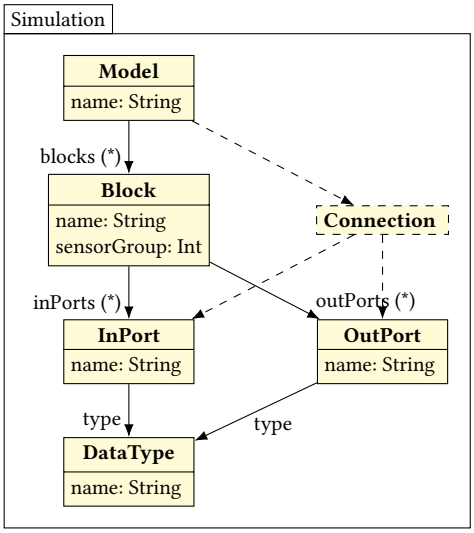

Figure 1: Example metamodel for the simulation of hardware components, consisting of connected blocks and ports. The references from and to the metaclass *Connection* are dashed, as they are not relevant for our example.

model, contains a number of *Sensors*, which have an *ID*, a *resolution*, and are part of a *sensorGroup*.

Notably, the system models for which we want to define and derive views consist of pairs, namely a blocks-and-ports and a sensor hardware model. This is, views project and combine their information to fulfill specific information needs. While, principally, such pairs (or, more generally, tuples) of models can be considered as a single model over a combined metamodel, the queries we address in this paper at least partially owe their complexity to the fact that they combine information from families of models into a single view. As an example, consider the system models and view presented in Figure 4. In the view, we see all blocks associated to sensors with a resolution greater than 16. Developers could use this view to analyze the blocks and check whether they support the high resolution data from the sensors.

4 Background

In this section, we recall TGGs (Section 4.1) and summarize which kind of model queries have already been translated into TGG rules by König et al. in [27] (Section 4.2).

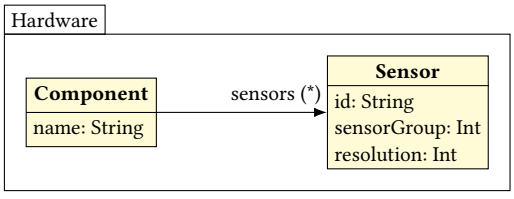

Figure 2: Example metamodel for sensor hardware.

4.1 Triple Graph Grammars

Triple graph grammars (TGGs) are an established formalism to specify the correspondence between models that are situated at different levels of abstraction [2, 32] and have already previously been used to specify views of models [3, 24]. In this section, we recall TGGs, the languages they define, and the operationalization of TGG rules, which is the basis for using TGGs for advanced model management tasks like model translation or synchronization. We introduce these notions formally here, where we need the corresponding notation in Section 6. Other notions are presented intuitively, and we provide references to the relevant literature for their formal definitions. For readers unfamiliar with TGGs, we use our running example in Appendix A to illustrate each introduced concept.

Variants of TGGs and their languages. We define triple graphs using the formalism suggested by Golas et al. [18] that slightly deviates from the original definition from Schürr [32]; however, we use a set theory-based definition (instead a morphism-based one as Golas et al. do).

A *triple graph* is a *typed graph* (i.e., it is possible to assign recurring *roles* to its elements) consisting of three distinguished parts being subgraphs of the triple graph: the *source*, the *target*, and the *correspondence graph*. Intuitively, the source graph captures a model in one modeling language (e.g., some design model of a system), the target graph captures a model in another modeling language (e.g., some abstracted view of the design model), and the correspondence graph serves as a structure to allow for tracing between both.

Definition 4.1 (Triple graph). A *triple graph* is a tuple (G, G_S, G_C, G_T), where G is a graph and G_S, G_C, G_T are subgraphs of G, referred to as *source*, *correspondence*, and *target part*, such that:

(1) The set of nodes V_G of G is a disjoint union of the sets of nodes $V_{G_S}, V_{G_C}, V_{G_T}$.
(2) Every edge $e \in E_G$ of G maximally belongs to one of the edge sets E_{G_S} or E_{G_T}; in particular, the edge set E_{G_C} is empty.
(3) Every edge $e \in E_G$ that neither belongs to E_{G_S} nor to E_{G_T} is directed from a node from V_{G_C} to either a node from V_{G_S} or from V_{G_T}.

To further enhance the expressiveness of the considered models, triple graphs can be *attributed*, i.e., nodes can carry values for various attributes. A general formalization of the attribution of graphs can be found in [11].

Triple graph grammars are a declarative, rule-based formalism that describe how two models from the source and the target domain co-evolve. This defines a correspondence-relation between pairs of models from the two domains by stipulating that two models are in correspondence when a triple graph can be derived, starting at the empty graph and using the rules of the given grammar, such that the one model is its source part and the other its target part. While the grammar defines correspondence on the level of models, the correspondence links in a triple graph define correspondence on the element level. TGG rules are *monotone*, i.e., they only create structure and do not delete. Intuitively, a TGG rule requires a certain pattern to exists, extends that pattern, but only does so if further specified forbidden pattern are not present. As advanced concepts, TGG rules can manipulate attribute values of graph nodes [5] and be equipped with so-called *negative application conditions (NACs)* that restrict the context in which a rule is allowed to be applied [4].

Definition 4.2 (TGG rule. Application). A *(TGG) rule* $\rho = (r\colon L \hookrightarrow R, NAC, C)$ consists of a *plain rule* r, a set NAC of negative application conditions, and a set C of *attribute constraints*. The plain rule r is an injective morphism $r\colon L \hookrightarrow R$ between triple graphs L – the *left-hand side (LHS)* – and R – the *right-hand side (RHS)* – of the rule; the LHS L of a rule is also referred to as *context* of the rule. The set NAC is a set of triple graphs such that L is a subgraph of each of them. The set C consists of Boolean expressions, regulating the attribute values appearing in the LHS L of the rule. A rule ρ is *applicable* to a triple graph G if there exists an injective morphism $m\colon L \hookrightarrow G$ that (i) cannot be extended to a morphism $q\colon N \hookrightarrow G$ for any of the graphs $N \in NAC$ and (ii) induces an evaluation on the attribute values of the LHS L such that every condition $c \in C$ is satisfied by it. *Application* of a rule then means to extend G by $R \setminus L$ (the node- and edge-wise difference of L and R) at the location indicated by m, obtaining a triple graph H as result; formally, this amounts to computing H as a pushout of m and r. Such an application, or *transformation step* is denoted via $G \Longrightarrow_{\rho,m} H$; a sequence of transformations via $G \Longrightarrow^* H$ or via $G \Longrightarrow^*_{\mathcal{R}} H$ if one wants to indicate that all transformation steps have been performed applying rules from a set $\mathcal{R}$.

A further feature that has been developed for TGGs is *multi-amalgamation*. Being established in the area of graph transformation [17], Leblebici et al. transferred it to TGGs [31]. Multi-amalgamation equips certain rule elements with a "for-each"-like syntax, enabling one to concisely express that a certain operation should be performed as often as possible. A *kernel rule* specifies elements that are to be matched or created (depending on which side of the rule they appear in) once, i.e., it is an ordinary TGG rule. Additional *multi-rules* extend this kernel rule by further context elements and elements to be created. Their application semantics is as follows: Given a match for the kernel rule, the kernel rule is applied exactly once at this match. The larger context of each multi-rule is matched as often as possible, extending the match of the kernel rule, and, for each such match, the elements declared to be created by the multi-rule are created at that match.

TGGs consist of sets of TGG rules and are used to define various languages. Two models from the source and the target modeling languages are considered to be consistent to each other when there exists a triple graph in the language of the given TGG such that the source and the target model are its source and target parts.

Definition 4.3 (Triple graph grammar. Language). A *triple graph grammar (TGG)* consists of a set of TGG rules $\mathcal{P}$. The *language* of a TGG is the set of triple graphs derivable via the given rules from the empty graph, i.e., $\mathcal{L}(\mathcal{P}) \coloneqq \{(H, H_S, H_C, H_T) \mid \emptyset \Longrightarrow^*_{\mathcal{P}} H\}$.

Importantly, all features of TGGs we considered (NACs, attribute constraints, and multi-amalgamation) increase the expressiveness of TGGs, i.e., allow one to define languages that cannot be defined using just plain rules [34].

Operationalization of TGG rules. TGGs as introduced so far describe how consistent models co-evolve; i.e., in our application scenario, they describe how to consistently co-evovle system models and views. The main usage scenario for TGGs, however, is not

Table 1: Overview of the basic query operators defined by König et al.

Name	Symbol	Arguments
Selection	$\sigma(P \Rightarrow Q)$	P Source pattern Q Target pattern
Filter	$\phi(p)$	p Filter predicate
Attribute Projection	$\pi(x \Rightarrow y)$ $\pi(y := f(\dots))$	x Source attribute y Target attribute f Projection function
Reference Projection	$\rho(R \Rightarrow S)$ $\rho_c(R \Rightarrow S)$	R Source reference pattern S Target reference pattern

the *construction* of correlated models but more challenging model management tasks like the *translation* of a given model into one of the second domain or *consistency restoration* after one or both previously corresponding models have been edited. For these tasks, TGG rules can be (automatically) *operationalized*. The crucial operationalized rule variant for this paper are so-called *forward rules*. A forward rule of a given TGG rule adapts that rule in the following way: The context elements of the original rule remain context elements of the forward rule. Elements that the original rule creates on the correspondence or the target side also remain to be created. Elements that the original rule creates on the source side, however, become context. Furthermore, to keep track of already translated elements, these formerly created elements get marked. This marking process can be formalized in various different ways, e.g., via translation attributes [21], via subrules [30], or via dedicated sub-triple graphs [25]. By and large symmetrically to the case of forward rules (except for the assignment of attribute values), *backward rules* can be derived from TGG rules. Instead of parsing structure on the source part of a triple graph and transferring it to the target part (as forward rules do), backward rules parse the target part and transfer the according structure to the source part. In our application case in this paper, the forward rules are of crucial importance: Given a model of a system, they can be used to compute a view of it. In the future, we also want to use forward as well as backward rules to propagate changes between models and views.

4.2 Operator-Based View Definition

Here, we give a brief overview of the model query operators defined by König et al. [27]. There, a view definition describes an asymmetric transformation between one or multiple source models and a projective view on these models. At the same time, the view definition describes a transformation to create the view type, i.e., the metamodel of the resulting views. A view definition consists of a set of model queries, which describe parts of the transformation. For that, a model query projects a fixed set of related elements of the source models to a fixed set of elements in the view.

To write view definitions, König et al. introduced an operator-based notation for model queries. We present an overview of the available operators in Table 1. A model query starts with a single selection operator, which projects the source elements to the view, and can contain an arbitrary number of other operators in addition, a center dot $\cdot$ indicating concatenation. For that, the selection operator σ takes a source pattern and a target pattern. The source pattern describes the model elements and their relations in the source models, while the target pattern describes the resulting elements in the view. Elements of the models and the view are specified using the name of their metaclass. König et al. restricted source patterns to directed paths, but we accept any well-typed graph.

In Query (1) we give a small example of a model query based on the two metamodels in Figs. 1 and 2. The relations in the source pattern can be explicit references, written as $\rightarrow_{foo}$ where *foo* is the name of the reference, or implicit in form of a join, written as $\bowtie_{bar}$ where *bar* is a boolean condition. In the example in Query (1), we join a *Block* with a *Sensor* and, from those, create a *Block* and a referenced *DebugInfo* in the view (instead of a join condition we only write the shared attribute as a shorthand). Joins between model elements behave similar to joins from relational algebra [9], but we will discuss some limitations in more detail in Section 5.2.

$$\begin{aligned}&\sigma(\text{Block} \bowtie_{\text{sensorGroup}} \text{Sensor} \Rightarrow \text{Block} \rightarrow_{\text{debug}} \text{DebugInfo}) \cdot \\ &\phi(\text{Block.resolution} > 16) \cdot \\ &\pi(\text{Sensor.resolution} \Rightarrow \text{DebugInfo.sensorRes}) \cdot \\ &\rho(\text{Block} \rightarrow_{\text{inPorts}} \text{InPort} \Rightarrow \text{Block} \rightarrow_{\text{in}} \text{Input})\end{aligned} \tag{1}$$

Additionally, there is the filter operator ϕ, which selects model elements based on a filter condition. The transformation described by the model query is only executed if the filter condition evaluates to true on the model elements described by the source pattern of the selection. In the example in Query (1), we only project blocks associated with sensor with a resolution greater 16.

The view elements can be further populated by projecting attributes and references from the source models. Attributes are projected using the attribute projection operator π, which either selects an attribute from the source elements or computes a new attribute. The first mode is written as $A.a \Rightarrow B.b$ where a is an attribute of the source element A and b is the resulting attribute in the target element B. We do this in the example in Query (1) to project the sensor attribute *resolution* to the *DebugInfo* element as attribute *sensorRes*. The second mode is written as $B.b := f(\dots)$ where f is a function on the attributes of the source elements.

Similar, references are projected using the reference projection operator ρ. While the source pattern of the selection operator can contain only 1-to-1 references to other model elements, the reference projection operator projects 1-to-n references to model elements, which are transformed separately. König et al. limited this to only one 1-to-n reference per source reference pattern. Instead, we define that the reference projection operator selects instances of the source reference pattern in the models. For each unique instance of the source reference pattern, an instance of the target reference pattern is created in the view. The source reference pattern may include elements from the source pattern of the query, while the target reference pattern creates exactly one reference from one of the elements in the target pattern. In the example in Query (1), we project the reference *inPorts* to the view, specify the view metaclass *Input* as its target, and change its name to *in*. A special case of the reference projection operator is containment projection, written as

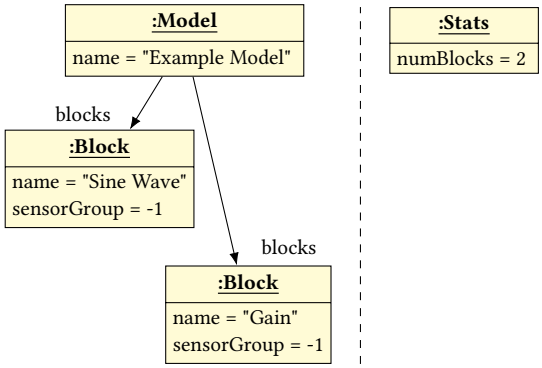

Figure 3: Part of a model (left) and a view (right) showing the aggregation over multiple referenced model elements.

ρ_c. In this case, the referenced elements are only transformed if the containing element is transformed.

5 Complex View Definition Operators

Beyond the basic operators recalled in Section 4.2, König et al. [27] encountered a number of limitations of their model query operators. Most of them exist because the necessary TGG would be complicated to construct. In this section, we examine three of them, namely *aggregation*, *cross-product joins*, and *overlapping queries*; in Section 6, we propose solutions using advanced TGG extensions.

5.1 Aggregation

With the operators presented in Section 4.2, we are able to project existing attributes, as well as calculate new attributes from the values of the attributes of the model elements in the source pattern of a query. However, as the source pattern has a constant size, we cannot calculate the value of an attribute from a set of model elements of unknown size, e.g., all elements referenced by an element in the source pattern. What we would like to have is an aggregation function, similar to aggregation in extensions of relational algebra [9]. For that, we introduce the aggregation operator α, which takes the application of an aggregation function as its first argument and the definition of a view attribute as its second argument. For the aggregation function we allow functions that incrementally build up the resulting value, i.e., functions that can be written as a fold over the aggregated elements. This is similar to how aggregation is handled in relational query languages. We show an example of this in Query (2) using *count* as an aggregation function to count the number of blocks in a model. In Fig. 3, we show an example model instance and the view resulting from applying the query in Query (2) to it. As the model references two blocks, the attribute *numBlocks* of the view element *Stats* is set to 2.

$$\begin{aligned} &\sigma(\text{Model} \Rightarrow \text{Stats}) \cdot \\ &\alpha(\mathit{count}(\text{Model} \rightarrow_{\text{blocks}} \text{Block}) \Rightarrow \text{Stats.numBlocks}) \end{aligned} \tag{2}$$

Generating a TGG for the aggregation is, however, difficult with traditional TGG formalisms. It is well-known that TGGs struggle to relate a structural encoding of information to an attribute-based one [1, Sect. 3.2]. We therefore consider this an interesting problem.

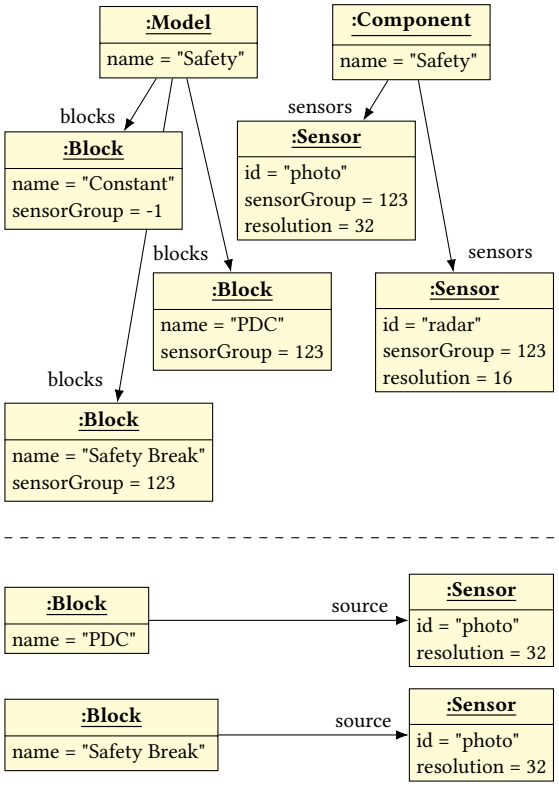

Figure 4: Part of two models (top) and a view (bottom) showing a conditional cross-product join of elements from the two models.

5.2 Cross-Product Joins

The selection operator σ, presented in Section 4.2, can be used to select elements related by explicit references or implicitly by joining two elements. For the latter, this is done using a join condition, e.g., $\bowtie_{\text{Model.name} = \text{Component.name}}$. However, König et al. [27] restrict the join conditions to 1-on-1 joins, i.e., a model element can only participate once in the same join. In general, of course, we want to be able to do full cross-product joins as well. In our example in Query (3), we want to join blocks with their associated sensors to present developers with a list of blocks and sensors to check, as motivated in Section 3. For that, we do not want to transform the elements individually, but instead produce a *Block* element as well as a *Sensor* element for each associated pair in the source models. We further restrict the selected pairs with a filter condition. In Fig. 4, we show the result of this join on a small example model. Note that for a more complex model, the same *Block* or *Sensor* element could occur multiple times, in different joined pairs, in the view.

$$\begin{aligned} &\sigma(\text{Block} \bowtie_{\text{sensorGroup}} \text{Sensor} \Rightarrow \\ &\qquad \text{Block} \rightarrow_{\text{source}} \text{Sensor}) \cdot \\ &\phi(\text{Sensor.resolution} > 16) \end{aligned} \tag{3}$$

5.3 Overlapping Queries

When generating a TGG from relational queries, König et al. [27] make the assumption that queries in a view definition do not overlap, i.e., that they do not reference the same source model elements. However, there certainly are cases where we would like to have overlapping queries. For example, we could show the same element with different representations, as done in projectional editing, use model elements to highlight constraint violations, or present summary information. We show queries Queries (4) and (6) as an example. Note that we assume that *InPort*, *OutPort*, and *DataType* elements are transformed separately with identical metaclass names and all attributes to simplify the example.

$$\sigma(\text{Block} \Rightarrow \text{InternalBlock}) \cdot \\ \phi(startsWith(\text{Block.name}, \text{“\_”})) \tag{4}$$

$$\sigma(\text{Block} \Rightarrow \text{SensorBlock}) \cdot \\ \phi(\text{Block.sensorGroup} > -1) \cdot \\ \rho(\text{Block} \rightarrow_{\text{outPorts}} \text{OutPort} \Rightarrow \text{SensorBlock} \rightarrow_{\text{out}} \text{OutPort}) \tag{5}$$

$$\sigma(\text{Model} \Rightarrow \text{ModelDiagnostics}) \cdot \\ \rho(\text{Model} \rightarrow_{\text{blocks}} \text{Block} \rightarrow_{\text{inPorts}} \text{InPort} \rightarrow_{\text{type}} \text{DataType} \\ \Rightarrow \\ \text{ModelDiagnostics} \rightarrow_{\text{types}} \text{DataType}) \cdot \\ \rho(\text{Model} \rightarrow_{\text{blocks}} \text{Block} \rightarrow_{\text{outPorts}} \text{OutPort} \rightarrow_{\text{type}} \text{DataType} \\ \Rightarrow \\ \text{ModelDiagnostics} \rightarrow_{\text{types}} \text{DataType}) \tag{6}$$

Queries (4) and (5) both select *Block* elements, however, Query (4) filters the result for internal blocks (which have names starting with an underscore), while Query (5) filters for blocks that have sensors associated to them. As there might be internal blocks that have sensors associated to them, a block from the source model might be transformed by none, either, or both of the queries.

Query (6) also selects *Block* elements, but in a source reference pattern. As the *inPorts* reference in the source reference pattern is used here exclusively, we could create a reference *types* in the view whenever a reference *inPorts* is found in the source model. Note that the *Block* element, from which the *inPorts* references originates, might or might not be transformed by another query.

Finally, Query (6) projects the *outPorts* reference to the *types* reference in the view type. As Query (5) also projects the *outPorts* reference, all parts of the source reference pattern in Query (6) might or might not be transformed by other queries.

Overlapping queries are difficult to handle when generating TGGs for the transformation. In common variants of TGGs, nodes are either context nodes or create nodes. By that, whenever a node in a rule might or might not be created by another rule, we would need to duplicate the rule with a context node in one and a create node in the other rule. As this can be the case for multiple nodes in a rule, we would require 2^n rules where n is the number of nodes with unique overlaps to other queries.

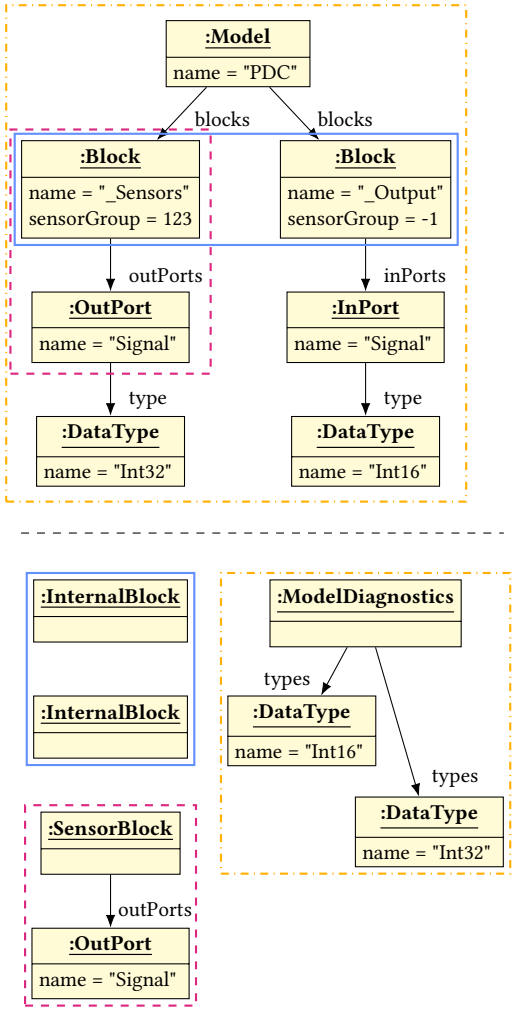

Figure 5: Part of a model (top) and a view (bottom) showing the result of the three overlapping Queries (4, blue, solid), (5, red, dashed), and (6, yellow, dashed and dotted).

6 Using and Extending Triple Graph Grammars

In this section, we sketch how we solve the problems introduced in the last section.

6.1 Aggregation

Support for aggregation has to be developed separately for each kind of aggregation function. For many basic aggregation functions, defining TGG rules that co-evolve models consistently capturing the aggregation function is possible, just using means that are well-established for attributed graphs, namely using rules to set attribute values according to operations supported by an underlying algebra [11]. The resulting rules, however, are such that they cannot always be operationalized in both directions, caused by the fact that regularly, the operations used to implement the aggregation are not invertible in a unique way. That is, we are able to operationalize such a TGG rule into a forward rule but not (uniquely)

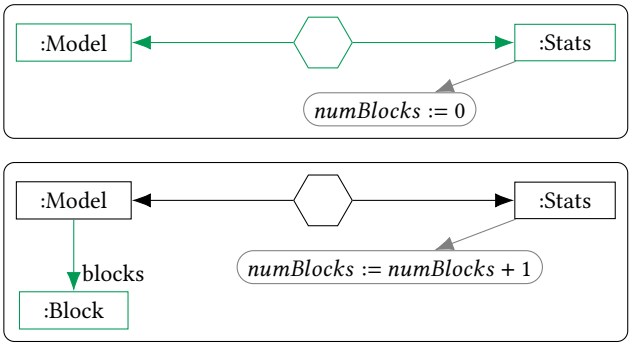

Figure 6: TGG rules for the aggregation defined in Query (2).

into a backward rule. For our application scenario, this does not cause a too serious problem, because we are mainly interested in constructing views (and not in adapting a model according to an edited view).

Given an aggregation like the one displayed in Query (2), our general approach to construct TGG rules from it is as follows.

(1) We translate the selection operation σ at the start of the query as already established in [27]. Additionally, we scan the second argument of the aggregation operator α for the view attribute defined there and create and initialize this attribute at the respective target node. The default value for initialization depends on the aggregation function. Additionally, depending on the aggregation function, the rule might create (but not yet initialize) a *helper attribute* at the correspondence node it creates; this shall serve to facilitate the computation of the value of the view attribute later on.[1]

(2) We translate the aggregation operation into a second TGG rule. The LHS, i.e., context of this rule is always the RHS of the rule constructed under 1., except that we omit the assignment of attribute values. Its RHS is specific to the selection operation σ and, in particular, the aggregation function to be translated. On the source part, the rule always creates the structure, or assigns the attribute value, that is declared in the source pattern P of the selection operation σ. On the target part, the rule always updates the view attribute according to the aggregation function. Additionally, depending on the aggregation function, the rule might need to update the helper attribute of the correspondence node of its LHS.

Example 6.1 (TGG rules for the count *function).* Figure 6 shows the two rules that arise for the example from Query (2), i.e., for the *count* function. The first rule creates the correspondence between the source and the target pattern defined in the selection operator of the query, creates the view attribute *numBlocks* and assigns to it the default value 0 (as suitable for the *count* function). The second rule creates a *Block* on the source part and increments the *numBlocks* attribute by 1.

[1]We position this helper attribute at the correspondence node because, usually, users do not interact with these and so we minimize diversion of users. It would also simply be possible to maintain these helper attributes as a separate data structure, but we opted for this more integrated approach.

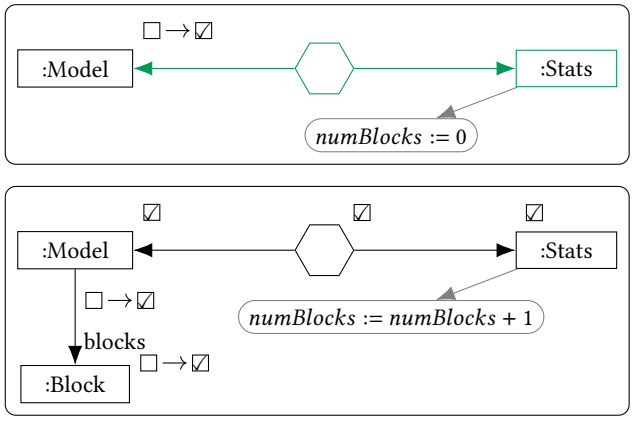

Figure 7: Forward rules of the TGG rules in Fig. 6.

In Appendix C we briefly sketch how we treat further common aggregation functions beyond the *count* function.

On the source part of a triple graph, the rules we derive for aggregation only match or create structure. We can therefore derive forward rules as usual, matching or marking as translated those elements on the source part instead of matching or creating them. On the correspondence and the target part, the elements are matched or created and attribute values assigned just as in the underlying TGG rule. Backward rules often cannot be derived, because, generally, aggregation functions are not (uniquely) invertible. While this reinforces the observation made in [1, Section 3.2] that TGGs are not well-equipped to synchronize in cases where information is represented as structure in one model but as attribute values in another, we do not see that this poses a problem to our application here, where we are foremost interested in just deriving the view, i.e., the more abstract model that might summarize structural information and represent it as a single numerical value, for instance.

Example 6.2. When forward operationalizing the TGG rules for the count function displayed in Fig. 6, we obtain the forward rules displayed in Fig. 7. The first rule marks an existing node of type *Model* as translated on the source part and creates a corresponding node of type *Stats* on the target part; moreover, it assigns the *numBlocks* attribute to 0. The second rule matches an already translated *Model* node and its corresponding *Stats* node on the target part, marks one of its previously untranslated *Blocks* as now translated and increments the corresponding *numBlocks* counter by 1.

6.2 Cross-Product Joins

TGGs have been designed to be able to express $m \times n$-relations between elements on the source and the target part [32]. Here, however, as introduced in Section 5.2 we are interested in a more general problem, namely in computing a *join* (i.e., an $m \times n$-relation between elements) on the source side and relating this join to a structure on the target part. To express the desired behavior compactly, we make use of multi-amalgamated TGG rules. Given a

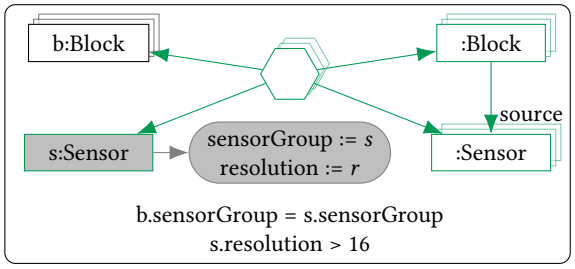

Figure 8: TGG rule for the left side of an inner join, as defined in Query (3).

selection query $\sigma(P \Rightarrow Q)$, where the source pattern $P = P_1 \bowtie_c P_2$ is defined as a join with join condition c, we derive two rules.

The first rule has as kernel rule a rule that just creates the source pattern P_1. Depending on the join condition c, the kernel rule might also create attributes for nodes of P_1 and assign these to variables. This kernel rule is extended by a single multi-rule, which matches the source pattern P_2 and creates the target pattern Q and a correspondence node that connects to each node of P_1, P_2 and Q. Moreover, the multi-rule is equipped with an attribute condition (having access to the variables used in the kernel rule) that ensures that it is only applicable when the join condition c is met. The second rule is completely analogous just that the roles of P_1 and P_2 are switched.

Example 6.3. Figure 8 shows the rule we derive for the left side of the join from Query (3). This rule ensures the completeness of the considered join by making use of multi-amalgamation: A newly created *Sensor* is joined with each already existing *Block* (of the same sensor group) and together these are put into correspondence with a pair of *Block* and *Sensor* on the target side. We show the rule for the right side of the join in Appendix B.

Operationalizing multi-rules to obtain forward and backward rules is treated in [31] and we can use these results without further adaptation. Intuitively, the forward rule we obtain from the rule in our example (Fig. 8) matches a yet untranslated *Sensor*, translates it and joins that *Sensor* to each already translated *Block*, correlating this pair to a newly created pair of *Sensor* and *Block* on the target part. We depict this rule (and also the forward rule derived for the right side of the join) in Appendix B.

6.3 Overlapping Queries: An Extended Skip Semantics

Kosiol et al. [26] have developed a skip semantics for graph transformation rules that we want to use to address the problem of overlapping queries. The main idea is that the actions of a rule are split into two classes: *mandatory* ones that have to be performed upon application of the rule and *potential* ones that are skipped if the outcome that the action would achieve is already present in the model. That is, for monotone rules as is the case for TGGs, a potential creation of an element is just skipped if a suitable element already exists in the model. This general idea can directly be transferred to the ordinary TGG rules that make up the grammar. However, we have to newly develop a concept of how to operationalize this skip semantics and ensure the termination of the resulting operationalized rules. It turns out that this is easiest when further extending the original skip semantics with counters that keep track of how often an element has been skipped.

In the following, we develop this idea and explain how we use it to translate overlapping queries into TGG rules. For this, we first introduce the concept of a *subrule*. Intuitively, in our case, a subrule comes without an application condition, matches the same context elements, but potentially creates less elements than its encompassing rule.

Definition 6.4 (Subrule of a TGG rule). Given a TGG rule $\rho = (r\colon L \hookrightarrow R, NAC, C)$, a *subrule* of ρ is a rule $\rho' = (r'\colon L' \hookrightarrow R')$ such that $L = L'$ and there is an injective morphism $i_R^{R'}\colon R' \hookrightarrow R$ such that $r = i_R^{R'} \circ r'$.

The intuition behind rules with skip semantics is the following. First, a subrule captures the mandatory actions; hence, the subrule is applied as is. A rule that encompasses the subrule captures the additional potential actions (only creations in our case). Before applying this larger rule, it is first checked which of the to be created elements can be considered as already existent. The according creation actions are then skipped and only the remaining ones performed. For this, we just reuse the definition of effect-oriented rules and transformations from [26]. However, extending [26], NACs are allowed to not just extend the LHS of the rule but also elements that are potentially to be created (because these might become context elements of the rule upon application). Moreover, the derivation of operationalized TGG rules with skip semantics that we sketch at the end of this section is a new contribution. Formally, the skip semantics rests on a factorization of the morphism that embeds the subrule into the ambient rule.

Definition 6.5 (TGG rule with skip semantics). A *TGG rule with skip semantics* is a pair of rules $\rho_{\text{skip}} = (\rho' = (r'\colon L \hookrightarrow R'), \rho = (r\colon L \hookrightarrow R, NAC, C))$, i.e., ρ' is a subrule of ρ, such that each graph $N \in NAC$ contains L, but maybe also nodes from $R \setminus R'$. Given a match $m\colon L \hookrightarrow G$ for the rule ρ, an *application* of ρ_{skip} is defined as follows.

(1) One searches for the largest (in terms of numbers of elements) factorization $i_{L_{\text{ind}}}^{L}\colon L \hookrightarrow L_{\text{ind}}$, $i_R^{L_{\text{ind}}}\colon L_{\text{ind}} \hookrightarrow R$ of r such that (i) L_{ind} does not contain any element from $R' \setminus L$ (i.e., no mandatory creations) and (ii) there exists an injective morphism $m_{\text{ind}}\colon L_{\text{ind}} \hookrightarrow G$ with $m_{\text{ind}} \circ i_{L_{\text{ind}}}^{L} = m$; this leads to an *induced rule* $i_R^{L_{\text{ind}}}\colon L_{\text{ind}} \hookrightarrow R$ with *induced match* $m_{\text{ind}}\colon L_{\text{ind}} \hookrightarrow G$.
(2) Every negative application condition $N \in NAC$ is checked: If N contains a skip node from $R \setminus L_{\text{ind}}$, N is just dropped. Otherwise, one computes $N' \coloneqq N \cup L_{\text{ind}}$, and N' becomes part of the negative application condition of the induced rule.
(3) Finally, one applies the induced rule $i_R^{L_{\text{ind}}}\colon L_{\text{ind}} \hookrightarrow R$ at the induced match m_{ind}, checking the newly computed negative application condition. If that is violated, the overall application is stopped.

Note that the application semantics is not deterministic as, in general, there might not exist a unique largest graph L_{ind} for factorizing r. In our applications, so far, we did not observe any problems

caused by that. We first describe the rules with skip semantics that we construct and later illustrate the concept using these rules.

Given a sequence of queries, we iteratively construct the following rules with skip semantics:

(1) For each query, we generate TGG rules as developed in [27] and recalled in Section 4.2 or introduced in Sections 5.1 and 5.2. This provides us with a complete set of operators defining consistency between model and view on the level of the individual queries; however, some queries might overlap.
(2) For each pair of rules (ρ_1, ρ_2) we calculate the intersection
$$R' := (R_1 \setminus L_1) \cap (R_2 \setminus L_2)$$
and extend ρ_1 and ρ_2 to the TGG rules with skip semantics
$$\rho_i^{\text{skip}} = (L_i \hookrightarrow (R_i \setminus R'), \rho_i).^2$$
That is, the elements commonly created by both rules get equipped with the skip semantics.
(3) To later ensure termination of the operationalized rules, we inspect the resulting rules with skip semantics to ensure that every such rule has a mandatory action on the source part, i.e., we search for rules $\rho^{\text{skip}} = (\rho' = (r' : L \hookrightarrow R'), \rho = (r : L \hookrightarrow R, C, NAC))$ where $L_S = R'_S$ (that is, on the source part, the LHS of the subrule coincides with the RHS of the second rule). For any such rule, we use the correspondence node that ρ creates between the source and target (reference) patterns of its underlying query, ensure that this is of a unique type, and add an additional NAC to ρ that forbids that the elements from the underlying source (reference) pattern are already connected to a correspondence node of that type.

Example 6.6. Figure 9 shows two of the rules we derive for Queries (4) to (6); the elements marked by "sk" are the potential creations allowed to be skipped. The two rules originally stem from the rules derived to translate the two reference projections from Query (6). Because both rules create *Blocks* and their incoming edges, these elements get equipped with the skip semantics. Moreover, the second rule also overlaps with the rule derived to translate the reference projection from Query (5); hence, also the *outPorts*-edge receives the skip semantics. In this way, the second rule ends up without any mandatory creation on its source part. We therefore ensure that the correspondence node that the rule creates has a unique type and add a NAC that forbids the concerned source elements to already be connected to such a correspondence node. Note that the referenced *Block* is an element with skip semantics. Thus, the NAC just gets dropped whenever the *Block* is actually created by an application of that rule, but gets evaluated whenever the *Block* is skipped, i.e., matched to an already existing *Block*. In this way, we prevent the rule from being applicable multiple times to the same set of *Model*, *Block* and *Port*.

Regarding non-determinism, as soon as a *Model* already has several *Blocks*, it is not unambiguous which of these is used to compute

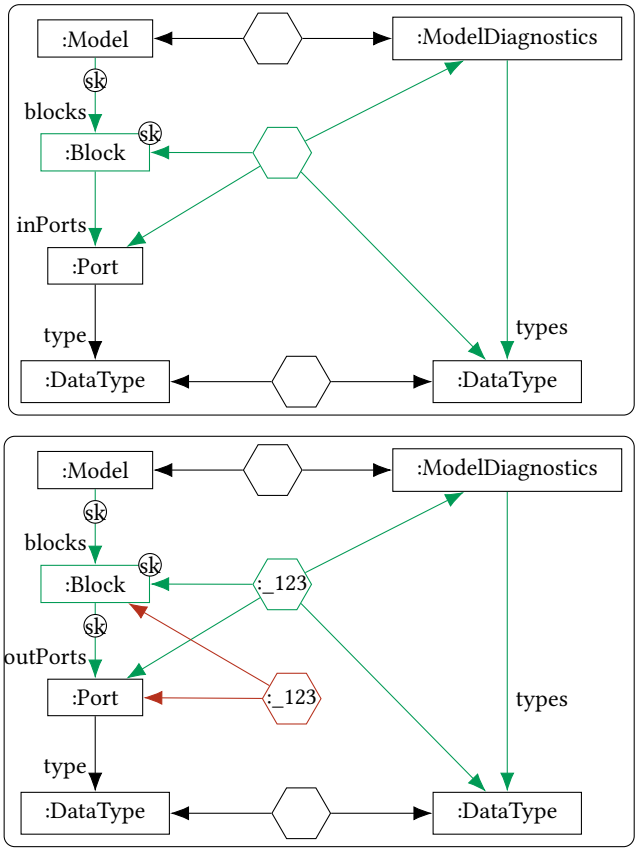

Figure 9: TGG rules for a query that overlaps with other queries, as defined in Query (6).

the induced rule. However, the resulting rules are isomorphic – just their matches differ. Importantly, each guarantees that a *Block* exists after its application. In our use case in this paper (deriving views from models), this non-determinism does not pose any problems, because we apply the forward rules obtained from those rules as long as possible. That is, the non-determinism affects the order in which *Blocks* are translated to the view but not the overall result. In the future, we intend to formally clarify the general conditions under which this holds.

From rules with skip semantics, we derive the following forward rules: Context elements and mandatory creations are operationalized as usual, i.e., on the source part these become elements that already need to be translated and elements that get translated by the rule application, respectively. On the correspondence and target parts, elements remain context elements and mandatory creations, respectively. Elements with skip semantics, in contrast, are treated as follows. On the correspondence and target parts, these elements keep their semantics, i.e., remain elements with skip semantics. On the source part, however, skip elements either translate a previously untranslated element or match an element that already has been translated. In our example, this leads to forward rules that match already translated pairs of corresponding *Model* and *ModelDiagnostics* nodes and *DataType* nodes, respectively, and, moreover, on the

[2] If one or both of the rules participating in the intersection are multi-amalgamated rules, we separately compute those intersections for the kernel and the multi-rule, resulting in a multi-amalgamated rule where the kernel as well as the multi-rule are allowed to be equipped with skip semantics. While we do not yet formalize this procedure in this paper, we are confident that this can be done.

source part match a path from *Model* to *DataType*, across a *Block* and a *Port*, where this path might already be translated partially; in that, matched untranslated elements get translated. The rules then create a correspondence node that connects the matched *Block* and *Port* to the matched *ModelDiagnostics* and *DataType* nodes on the target part and introduces a *types*-edge between the latter two. These rules are depicted in Appendix B.

7 Conclusion

In this paper, we continue the work of König et al. [27] of providing a formal semantics to the NeoJoin view definition language by translating NeoJoin queries into TGG rules. We address queries that had been omitted in [27] for their complexity. We find that – by and large – TGGs are well-suited for our task; however, we need to make use of advanced features of TGGs like negative application conditions, attribution and attribute conditions, and multi-amalgamation. As central technical contribution, to be able to deal with overlapping queries, we introduce a *skip semantics* for TGG rules, based on previous work by Kosiol et al. [26].

In the future, there are two avenues to continue our work. First, TGGs have regularly been used to obtain formal guarantees for model management processes (like model translation or synchronization) based on them [2, 16, 22, 25]. Based on this, we intend to work out important properties of the views we derive. Given our experience with and existing research on TGGs, we are confident that we will be able to prove the following (or at least find criteria under which the following holds): (i) The rules for aggregation behave correctly, i.e., aggregate attribute values or count elements as one would intuitively expect. (ii) The rules for joins behave correctly, i.e., construct the complete join. (iii) The rules for overlapping queries behave correctly, i.e., each element is constructed exactly once. (iv) The set of forward rules derived from all constructed rules is *confluent*, i.e., using them to actually compute views can be done by applying the set of rules as long as possible in arbitrary order. (v) User edits of a model or a view (except for manipulating aggregated attribute values) can be incrementally synchronized in either direction. The mentioned criteria could include limiting the expressiveness of the described operators, such as restricting the source patterns of the selection operator. While we do not yet prove results of this kind in this paper, developing a formal semantics as we do here makes such results tractable in the first place.

Second, and more practically, we intend to implement the concepts developed in this paper and want to extend the derivation of views, which we develop in this paper, by also synchronizing between models and views after one or both of them have been altered. Recent work on automatically generating repair operators for TGGs [14–16, 25] provides a solid foundation to achieve this in an incremental and information-preserving fashion.

Acknowledgments

This work builds on discussions from the NII Shonan Meeting No. 231 on Bidirectional Transformations: Foundations and Applications (bx). This work was supported by funding from the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF) and funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) — SFB 1608 — 501798263. We would like to thank Erik Burger for his helpful comments on an earlier version of this paper.

References

[1] Anthony Anjorin and Thomas Buchmann. 2026. On the Practical Expressiveness of Triple Graph Grammars. In *Proceedings of the 14th International Conference on Model-Based Software and Systems Engineering, MODELSWARD 2026, Marbella, Spain, March 7–9, 2026*, Federico Ciccozzi, Luís Ferreira Pires, and Francis Bordeleau (Eds.). SCITEPRESS, 410–417. doi:10.5220/0014448100004058

[2] Anthony Anjorin, Erhan Leblebici, and Andy Schürr. 2015. 20 Years of Triple Graph Grammars: A Roadmap for Future Research. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 73 (2015). doi:10.14279/TUJ.ECEASST.73.1031

[3] Anthony Anjorin, Sebastian Rose, Frederik Deckwerth, and Andy Schürr. 2014. Efficient Model Synchronization with View Triple Graph Grammars. In *Modelling Foundations and Applications – 10th European Conference, ECMFA@STAF 2014, York, UK, July 21–25, 2014. Proceedings (Lecture Notes in Computer Science)*, Jordi Cabot and Julia Rubin (Eds.). Springer, 1–17. doi:10.1007/978-3-319-09195-2_1

[4] Anthony Anjorin, Andy Schürr, and Gabriele Taentzer. 2012. Construction of Integrity Preserving Triple Graph Grammars. In *Graph Transformations – 6th International Conference, ICGT 2012, Bremen, Germany, September 24–29, 2012. Proceedings (Lecture Notes in Computer Science)*, Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg (Eds.). Springer, 356–370. doi:10.1007/978-3-642-33654-6_24

[5] Anthony Anjorin, Gergely Varró, and Andy Schürr. 2012. Complex Attribute Manipulation in TGGs with Constraint-Based Programming Techniques. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 49 (2012). doi:10.14279/TUJ.ECEASST.49.707

[6] Dominique Blouin, Alain Plantec, Pierre Dissaux, Frank Singhoff, and Jean-Philippe Diguet. 2014. Synchronization of Models of Rich Languages with Triple Graph Grammars: An Experience Report. In *Theory and Practice of Model Transformations – 7th International Conference, ICMT@STAF 2014, York, UK, July 21–22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8568)*, Davide Di Ruscio and Dániel Varró (Eds.). Springer, 106–121. doi:10.1007/978-3-319-08789-4_8

[7] Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. 2019. A feature-based survey of model view approaches. *Software & Systems Modeling* 18, 3 (June 2019), 1931–1952. doi:10.1007/s10270-017-0622-9

[8] Thomas Buchmann and Bernhard Westfechtel. 2016. Using triple graph grammars to realise incremental round-trip engineering. *IET Softw.* 10, 6 (2016), 173–181. doi:10.1049/IET-SEN.2015.0125

[9] E. F. Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (June 1970), 377–387. doi:10.1145/362384.362685

[10] Csaba Debreceni, Ákos Horváth, Ábel Hegedüs, Zoltán Ujhelyi, István Ráth, and Dániel Varró. 2014. Query-driven incremental synchronization of view models. In *Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO '14)*. Association for Computing Machinery, New York, NY, USA, 31–38. doi:10.1145/2631675.2631677

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

[12] Kevin Feichtinger, Kristof Meixner, Felix Rinker, István Koren, Holger Eichelberger, Tonja Heinemann, Jörg Holtmann, Marco Konersmann, Judith Michael, Eva-Maria Neumann, Jérôme Pfeiffer, Rick Rabiser, Matthias Riebisch, and Klaus Schmid. 2022. Industry Voices on Software Engineering Challenges in Cyber-Physical Production Systems Engineering. In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*. 1–8. doi:10.1109/ETFA52439.2022.9921568

[13] Charles L. Forgy. 1982. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19, 1 (Sept. 1982), 17–37. doi:10.1016/0004-3702(82)90020-0

[14] Lars Fritsche, Jens Kosiol, Alexander Lauer, Adrian Möller, and Andy Schürr. 2024. Advanced Model Consistency Restoration with Higher-Order Short-Cut Rules. *Log. Methods Comput. Sci.* 20, 3 (2024). doi:10.46298/LMCS-20(3:25)2024

[15] Lars Fritsche, Jens Kosiol, Adrian Möller, Andy Schürr, and Gabriele Taentzer. 2020. A precedence-driven approach for concurrent model synchronization scenarios using triple graph grammars. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual*

Event, USA, November 16–17, 2020, Ralf Lämmel, Laurence Tratt, and Juan de Lara (Eds.). ACM, 39–55. doi:10.1145/3426425.3426931

[16] Lars Fritsche, Jens Kosiol, Andy Schürr, and Gabriele Taentzer. 2021. Avoiding unnecessary information loss: correct and efficient model synchronization based on triple graph grammars. *Int. J. Softw. Tools Technol. Transf.* 23, 3 (2021), 335–368. doi:10.1007/S10009-020-00588-7

[17] Ulrike Golas, Hartmut Ehrig, and Annegret Habel. 2010. Multi-Amalgamation in Adhesive Categories. In *Graph Transformations – 5th International Conference, ICGT 2010, Enschede, The Netherlands, September 27 – October 2, 2010. Proceedings (Lecture Notes in Computer Science)*, Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr (Eds.). Springer, 346–361. doi:10.1007/978-3-642-15928-2_23

[18] Ulrike Golas, Leen Lambers, Hartmut Ehrig, and Holger Giese. 2012. Toward Bridging the Gap between Formal Foundations and Current Practice for Triple Graph Grammars – Flexible Relations between Source and Target Elements. In *Graph Transformations – 6th International Conference, ICGT 2012, Bremen, Germany, September 24–29, 2012. Proceedings (Lecture Notes in Computer Science)*, Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg (Eds.). Springer, 141–155. doi:10.1007/978-3-642-33654-6_10

[19] Thomas Goldschmidt, Steffen Becker, and Erik Burger. 2012. Towards a Tool-Oriented Taxonomy of View-Based Modelling. In *Proceedings of the Modellierung 2012 (GI-Edition – Lecture Notes in Informatics (LNI), Vol. P-201)*, Elmar J. Sinz and A. Schürr (Eds.). Gesellschaft für Informatik (GI), Bonn, Germany, 59–74. https://dl.gi.de/handle/20.500.12116/18148

[20] Fahad R. Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guerin, and Christophe Guychard. 2016. Addressing modularity for heterogeneous multi-model systems using model federation. In *Companion Proceedings of the 15th International Conference on Modularity (MODULARITY Companion 2016)*. Association for Computing Machinery, New York, NY, USA, 206–211. doi:10.1145/2892664.2892701

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

[22] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, Yingfei Xiong, Susann Gottmann, and Thomas Engel. 2015. Model synchronization based on triple graph grammars: correctness, completeness and invertibility. *Softw. Syst. Model.* 14, 1 (2015), 241–269. doi:10.1007/S10270-012-0309-1

[23] Frank Hermann, Susann Gottmann, Nico Nachtigall, Hartmut Ehrig, Benjamin Braatz, Gianluigi Morelli, Alain Pierre, Thomas Engel, and Claudia Ermel. 2014. Triple Graph Grammars in the Large for Translating Satellite Procedures. In *Theory and Practice of Model Transformations – 7th International Conference, ICMT@STAF 2014, York, UK, July 21–22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8568)*, Davide Di Ruscio and Dániel Varró (Eds.). Springer, 122–137. doi:10.1007/978-3-319-08789-4_9

[24] Johannes Jakob, Alexander Königs, and Andy Schürr. 2006. Non-materialized Model View Specification with Triple Graph Grammars. In *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17–23, 2006, Proceedings (Lecture Notes in Computer Science)*, Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg (Eds.). Springer, 321–335. doi:10.1007/11841883_23

[25] Jens Kosiol. 2022. *Formal Foundations for Information-Preserving Model Synchronization Processes Based on Triple Graph Grammars.* Ph. D. Dissertation. University of Marburg, Germany. https://archiv.ub.uni-marburg.de/diss/z2022/0224

[26] Jens Kosiol, Daniel Strüber, Gabriele Taentzer, and Steffen Zschaler. 2023. Finding the Right Way to Rome: Effect-Oriented Graph Transformation. In *Graph Transformation*, Maribel Fernández and Christopher M. Poskitt (Eds.). Springer Nature Switzerland, Cham, 43–63. doi:10.1007/978-3-031-36709-0_3

[27] Lars König, Daniel Ritz, and Erik Burger. 2025. Transforming Relational Model Queries to Triple Graph Grammars. In *2025 ACM/IEEE 28th International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 752–761. doi:10.1109/MODELS-C68889.2025.00098

[28] Lars König, Tobias Stickling, and Erik Burger. 2025. Towards dynamic views on heterogeneous models -– the NeoJoin view definition language. In *Joint Proceedings of the STAF 2025 Workshops: OCL, OOPSLE, LLM4SE, ICMM, AgileMDE, AI4DPS, and TTC co-located with the International Conference on Software Technologies: Applications and Foundations (STAF 2025)*. doi:10.5445/IR/1000188774

[29] Lars König, Tobias Stickling, Alexander Kocher, Hüseyin Kemâl Çakmak, Erik Burger, Veit Hagenmeyer, Anne Koziolek, and Ralf Reussner. 2026. Language Design of the NeoJoin View Definition Language. *The Journal of Object Technology* 25, 3 (2026), 1. doi:10.5381/jot.2026.25.3.a1

[30] Erhan Leblebici, Anthony Anjorin, Lars Fritsche, Gergely Varró, and Andy Schürr. 2017. Leveraging Incremental Pattern Matching Techniques for Model Synchronisation. In *Graph Transformation – 10th International Conference, ICGT 2017, Held as Part of STAF 2017, Marburg, Germany, July 18–19, 2017, Proceedings (Lecture Notes in Computer Science)*, Juan de Lara and Detlef Plump (Eds.). Springer, 179–195. doi:10.1007/978-3-319-61470-0_11

[31] Erhan Leblebici, Anthony Anjorin, Andy Schürr, and Gabriele Taentzer. 2017. Multi-amalgamated triple graph grammars: Formal foundation and application to visual language translation. *J. Vis. Lang. Comput.* 42 (2017), 99–121. doi:10.1016/J.JVLC.2016.03.001

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

[33] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. 2015. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming* 98 (Feb. 2015), 80–99. doi:10.1016/j.scico.2014.01.004

[34] Nils Weidmann, Robin Oppermann, and Patrick Robrecht. 2019. A feature-based classification of triple graph grammar variants. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20–22, 2019*, Oscar Nierstrasz, Jeff Gray, and Bruno C. d. S. Oliveira (Eds.). ACM, 1–14. doi:10.1145/3357766.3359529

A Examples for Notions from Triple Graph Grammars

In this section, we use our running example to comprehensively illustrate TGGs and their established extensions as briefly introduced in Section 4.1. We begin with an example for triple graphs.

Example A.1 (Triple graphs). Figures 3 to 5 can be interpreted as showing triple graphs in a simplified fashion. The part above or left of the dashed line is always the source part of the triple graph, the part below or right of it the target part. The source part is a system model defined over a combination of the two metamodels introduced in Section 3, the target part is a view of that model. We omit the correspondence parts because these would also be hidden from a user. Formally, for example, in Fig. 3 there also exists a correspondence node that has two outgoing edges leading to the *Model* and the *Stats* nodes, respectively. Note that all examples make use of typing and attribution of graphs.

The next example illustrates TGG rules.

Example A.2 (TGG rules). Figures 6, 8 and 9 depict various TGG rules, using a simple graphical syntax. Black parts indicate the LHS's of rules, i.e., the context elements that have to be matched to apply a rule; an empty LHS indicates that a rule is always applicable. Green parts are the elements from $R \setminus L$, i.e., the elements that are to be created. The rectangular elements on the left side of each rule always belong to the source parts of the graphs defining the rule, rectangular elements on the right to its target parts, and hexagon shaped elements are correspondence nodes. Statements in rounded, gray boxes define variable assignments (formally, these belong to the RHS of a rule and express that an attribution edge of the type indicated by the left part of the assignment is created that points to the attribute value indicated by the expression in the right part of the assignment). Thus, the second rule in Fig. 6 declares to match a *Model*- and a *Stats*-node that are in correspondence and to create a *Block*-node for the *Model* and increment the *numBlocks* counter by one. The equations under the two rules from Fig. 8 are attribute conditions, expressing that the rules are only applicable to *Blocks* and *Sensors* of the same sensor group and to *Sensors* of high enough resolution. We later explain the gray fill and shadowed boxes appearing in the rules from this figure. Finally, the red elements appearing in the second rule from Fig. 9 constitute a *negative application condition* (NAC). The NAC forbids this rule to be applied to a pair of *Block* and *Port* nodes that are already connected via an appropriately typed correspondence node. Again, we later explain the annotation of created elements via "sk" appearing in this figure.

The following example illustrates multi-amalgamated TGG rules.

Example A.3 (Multi-amalgamated TGG rules). Figure 8 displays two rules making use of multi-amalgamation. The elements filled gray belong to the kernel rule, meaning that they are matched or created (depending on whether they belong to the LHS or the RHS of the rule) once. Elements with shadows belong to the multi-rule, meaning that they are matched as often as possible (in case they belong to the LHS) or created once per match of the multi-rule (in case they belong to the RHS). The attribute condition is evaluated for the multi-rule. Summarizing, the first rule in Fig. 8 creates one sensor on the source side, sets is sensor group and resolution (according to user provided or default values) and for each *Block* existing on the source side and having a sensor group equal to the one of the created *Sensor*, it creates one *Block*, *Sensor* and reference between them on the target side and connects all four nodes via a newly created correspondence node.

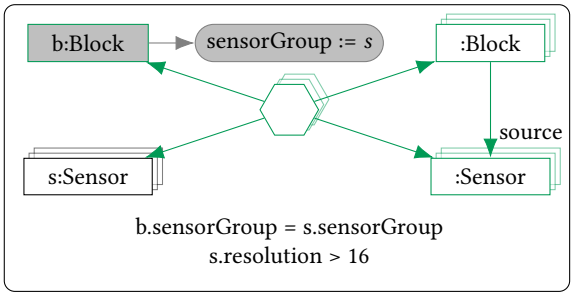

Figure 10: TGG rule for the right side of an inner join, as defined in Query (3).

Finally, we illustrate operationalized and, in particular, forward rules.

Example A.4 (Operationalized TGG rules). Figures 7, 11 and 12 show the *forward rules* that are derived from the TGG rules displayed in Figs. 6, 8 and 9. That an element is annotated with a check mark means that it was a context element of the underlying TGG rule and now is only allowed to be matched to elements that already have been "checked", i.e., have already been translated by the previous application of a forward rule. (In that, elements that have been created by a forward rule on the correspondence or the target side are interpreted as "checked"). If an element is annotated with an arrow transferring an empty box to a checked one, this means that this element is a source element that was to be created by the underlying TGG rule. Now, it is only allowed to be matched to source elements that have not yet been "checked" (translated) and checks such an element. In this way, forward rules can be used to translate a source model to a target model (also building the correspondence structure in between). That is, in our application case, forward rules can be used to compute views from (families of) models. The "checking" of elements that is necessary to keep track of which elements already have been translated can be formalized (and implemented) in various different ways.

B Examples for Additional and Operationalized Rules

In this section, we show rules that we explain but do not display in the main part of our paper. For the aggregation example, all constructed TGG rules as well as their derived forward rules are displayed in the paper (Figs. 6 and 7).

In Fig. 10, we show the TGG rule for the right side of an inner join, in addition to the TGG rule for the left side of an inner join, which we showed in Fig. 8.

Furthermore, Fig. 11 shows both forward rules we derive from these two TGG rules. As briefly explained at the end of Section 6.2, these forward rules translate a *Sensor* or *Block*, respectively, by

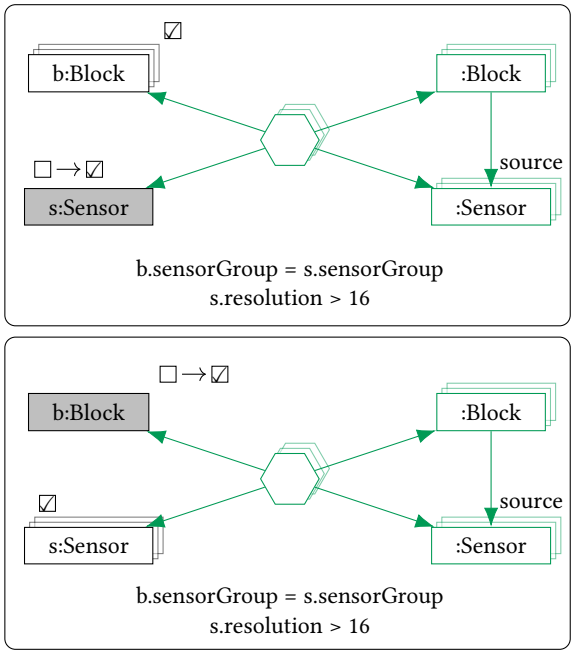

Figure 11: Forward operationalization of the multi rules for an inner join, as defined in Fig. 8.

joining it with each previously translated *Block* or *Sensor*, respectively, by creating the required target pattern (once per pair of *Block* and *Sensor*) and connecting everything via a newly created correspondence node.

Finally, Fig. 12 shows the two forward rules we derive from the TGG rules with skip semantics we introduce for the case of overlapping queries (Fig. 9); we briefly explained these rules at the end of Section 6.3. The notation ☑ / □→☑ indicates that an element is either allowed to be matched to an already translated or to a yet untranslated element. This means, formally, these rules depict rule schemata and encode several rules. This reflects the fact that the underlying TGG rules can, for example, either match or create a node of type *Block*.

C Details on the Treatment of Aggregation

Table 2 provides an overview of how we treat a range of common aggregation functions. Note that, whereas *Count* counts the number of graph elements of a certain type, the other aggregation functions read out some numerical attribute and perform a computation based on that. Depending on the kind of computation, we have to maintain a helper attribute that, in the worst case, keeps track of all attribute values encountered so far (e.g., in the cases of *Median* and *Mode*). The *Count* function is the only one that is uniquely invertible: here, the view attribute contains all information necessary to reconstruct the source structure from it. From an average, however, one can neither derive from how many nor from which exact values it is computed. For the other aggregation functions this is typical.

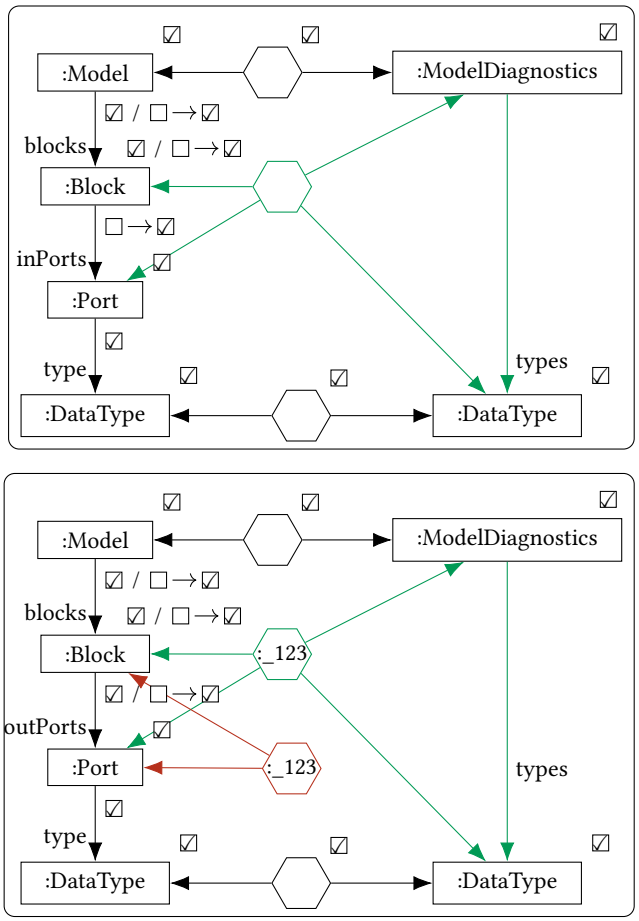

Figure 12: Forward operationalization of the TGGs rules for overlapping queries with skip semantics, as defined in Fig. 9.

Table 2: Overview of the translation of aggregation functions; *ra* refers to the attribute read out by the source pattern (if applicable), *ha* to the helper attribute, and *va* to the view attribute.

Function	**Type of** *ha*	**Default** *ha*	**Update** *ha*	**Default** *va*	**Update** *va*	Invertible
Average	Integer	0	$ha \coloneqq ha + 1$	0	$va \coloneqq \frac{(ha-1)\times va+ra}{ha}$	no
Count	—	—	—	0	$va \coloneqq va + 1$	yes
Max	—	—	—	$-\infty$	$va \coloneqq \max(va, ra)$	no
Median	Multiset	$\emptyset$	$ha \coloneqq ha \cup \{ra\}$	not set	$va \coloneqq \text{Median}(ha)$	no
Mode	Multiset	$\emptyset$	$ha \coloneqq ha \cup \{ra\}$	not set	$va \coloneqq \text{Mode}(ha)$	no
Range	Pair of numbers	$(\infty, -\infty)$	$ha = (\min(ha_0, ra), \max(ha_1, ra))$	$-\infty$	$va \coloneqq ha_1 - ha_0$	no
Sum	—	—	—	0	$va \coloneqq va + ra$	no