Transforming Relational Model Queries to Triple Graph Grammars

Lars König, Daniel Ritz, Erik Burger

KASTEL – Institute of Information Security and Dependability
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

***Abstract*—Views are an important part of model-driven development processes, as they allow developers to work on abstractions of potentially complex system models. The definition of model-view transformations is, however, difficult, especially if bidirectional synchronization between models and views is required. In this work, we present a relational operator model for specifying queries on heterogeneous models, as well as a transformation of our operators to triple graph grammars. View definition approaches can use our operator model as transformation backend and leverage the inherently bidirectional and incremental transformation operationalization of triple graph grammars. We further present an initial evaluation using a prototype implementation of our operator model and transformation, and discuss possibilities to extend the operator model.**

***Index Terms*—relational operators, triple graph grammars, view-based development, model transformation**

I. Introduction

In model-driven development, systems are described by a variety of heterogeneous models, conforming to different meta-models. With an increasing complexity of the systems under development, such as cyber-physical systems, developers struggle to have an overview of the models and, consequently, the system under development. This results in inefficient development processes and inconsistencies in the models, which increase development time and cost, and can even hinder the realization of a system entirely [6]. View-based development aims to solve this by providing developers with views specific to their role and task. Using an appropriate representation of the relevant information is intended to increase the efficiency of developers. Having views on shared underlying models can further improve the efficiency of development processes of teams, as the indirection through the views makes it easier to preserve the consistency of the underlying models [9].

Existing view definition languages, however, often fail to provide adequate support for agile, model-driven development processes [11]. To support developers in using appropriate views to work on the models of a system, we believe a view definition language should enable on-the-fly definition and generation of views while still allowing developers to commit their work to the underlying models. In short, we state the following requirements for view definition languages:

- A relational query language for the flexible definition of view types and view generation transformations
- Bidirectional and incremental transformation between models and views
- Support for heterogeneous models, conforming to different meta-models

In our paper, we present a relational operator model for queries on heterogeneous models, as well as a transformation from our operator model to triple graph grammars [16]. With our operator model, we offer an expressive backend for future view definition languages, as described in the work of König et al. [11]. To support bidirectional and incremental transformations between the underlying models and the defined views, we transform the queries to triple graph grammars and leverage their established operationalization. We further present a prototype implementation of both the operator model and the transformation to triple graph grammars, as well as an initial evaluation on two small cases. Both are included in the replication package [10] for this paper. Finally, we discuss future research questions to extend the expressiveness of our operator model.

II. Foundations

A. View-based Development

In model-driven development processes, models are the central development artifacts, which describe different parts and aspects of a system on different abstraction levels [15]. Abstractions are a crucial feature of model-driven development, as they reduce the complexity of models for developers for a specific set of tasks. We call models that serve as abstractions of a system from a specific viewpoint, i.e., for specific developers and specific tasks, *views* [8]. When views describe the same system from different viewpoints, they may contain overlapping information, which can be the cause of consistencies between the views. Inconsistencies increase the development cost and might even hinder the realization of the developed system [6]. There are various approaches for preserving consistency between views, such as projective views, which are generated on demand from an underlying repository [1]. In model-driven development processes with established tools and languages, this underlying repository

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.

is pragmatically constructed from existing models and meta-models [9]. Projective views can then be created by deriving information from multiple, heterogeneous models conforming to different meta-models. The content and generation of views, i.e., its meta-model and model-view transformations, can be defined using view definition languages, which offer different paradigms and features [2]. To lower the effort of defining views, we focus on view definition languages with a declarative, relational paradigm. In development processes with multiple developers using different views, bidirectional and incremental transformations between the models and the views are required.

B. Triple Graph Grammars

Triple graph grammars (TGGs) [16] are a formalism for graph construction rules, which can be used to describe model transformations. TGGs consist of triple graph rules, which describe combined construction steps in two graphs connected by a correspondence graph between them. To transform a source model to a target model or vice versa, both represented as graphs, TGGs can be operationalized to forward and backward transformations. In addition, they can be used to check the consistency of two models or to synchronize models after changes have been made to them, also called model integration. Triple graph rules are often expressed in their shorthand graphical syntax, consisting of black context nodes and edges and green nodes and edges. The context nodes and edges must be present in the graph for a rule to be applicable, while the green nodes and edges are added when applying a rule. An example of a triple graph rule is shown in Figure 8. In addition to the basic graph construction rules, various extensions to TGGs have been proposed [19]. Notable extensions include attribute conditions [12], which relate the attributes of nodes in a triple graph rule, negative application conditions, which prevent rule application when certain nodes exist, and multi-amalgamation. Multi-amalgamation [13] combines a single kernel rule with multi-rules that can be applied multiple times to, e.g., relate a created node with a dynamic number of existing nodes. As our prototype is based on eMoflon::Neo, we will use the same TGG flavor as Weidmann and Anjorin [18].

C. Relational Algebra

Relational databases describe entities and their relationships [17]. Databases consist of tables, in which the rows represent entities and the columns represent their attributes. Relationships between entities are encoded as related attribute values, which act as primary keys for the referenced entity and foreign keys for the referencing entity. Executing queries on their tables is a key feature of databases, with SQL as a common example of a query language. The operators available in database query languages, such as SQL, are based on relational algebra theory [5]. The fundamental operators of a relational algebra are *projection*, *join*, and *filter*. Extensions to the set of operators include, e.g., the *aggregation* of attribute values over groups of entities.

TABLE I
OVERVIEW OF OUR QUERY OPERATORS.

Name	Symbol	Arguments
Selection	$\sigma(P \Rightarrow Q)$	P Source pattern Q Target pattern
Filter	$\phi(p)$	p Filter predicate
Attribute Projection	$\pi(x \Rightarrow y)$ $\pi(y := f(x))$	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

III. Relational Operators

In this section, we define and describe the relational operators of our query language. An overview of the available operators in our language is shown in Table I. A query $q \in Q$ consists of exactly one *Selection* operator σ and may contain zero, one, or multiple *Attribute Projection* and *Reference Projection* operators π and ρ, as well as zero, one, or multiple *Filter* operators ϕ, as shown in (1). The operators in a query are concatenated using a center dot $\cdot$. The order of the operators in a query is irrelevant.

Note that the operators are not functions but syntax constructs in our query language. As such, they may also define a syntax for their arguments, as indicated in the *Symbol* column of Table I. A more detailed description of the operator arguments and their syntax is given in the following subsections.

$$q := \sigma \cdot \pi_0 \cdot \ldots \cdot \pi_n \cdot \rho_0 \cdot \ldots \cdot \rho_m \cdot \phi_0 \cdot \ldots \cdot \phi_l \tag{1}$$

Variables representing instances are introduced by the selection operator σ and can be used in the arguments of all other operators. The general syntax for introducing a variable for an instance of a meta-class C is $C\,var$, where var is the name of the variable. In this paper, we use the simplified syntax C, which introduces a variable with the same name as the class. In the arguments of the selection operator σ, we refer to meta-model elements, i.e., meta-classes, by their name, either fully qualified with the name of the meta-model or unqualified if the name is unique. For the other operators, we refer to attributes using a variable introduced in the selection operator of the query and the name of the attribute in the form $var.attr$, where $attr$ is the name of an attribute of a meta-class C, given that var is a variable for an instance of C.

$$\begin{aligned} \Gamma_M &: Q_M \to \mathscr{M} \\ \gamma_{M,V} &: Q_M \times M_0 \times \cdots \times M_n \to V \end{aligned} \tag{2}$$

A query $q \in Q_M$ is defined on a finite set of meta-models $M \subset \mathscr{M}$ with size n conforming to a common meta-meta-model $\mathscr{M}$. We refer to the meta-models in M as M_i with $i \in [1 \ldots n]$. From a query $q \in Q_M$, we can generate a view type $V \in \mathscr{M}$ on the meta-models in M. Furthermore, the query can also be used to transform instance models $m_i \in M_i$, conforming to the meta-models $M_i \in M$ on which the query is defined, to a view

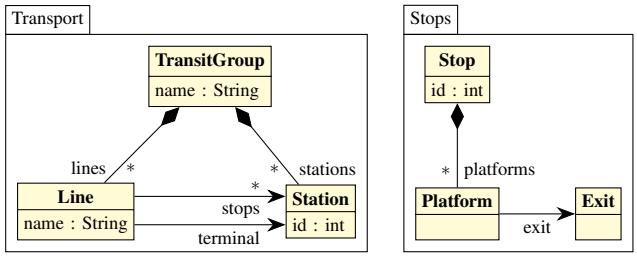

Fig. 1. Example meta-models *Transport* (left) and *Stops* (right).

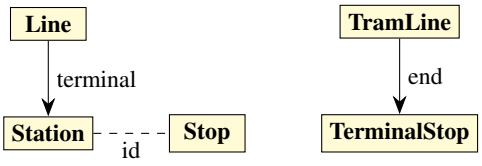

Fig. 2. Illustration of the source (left) and target (right) patterns for the example query in (3). The dashed line represents a join between two classes. The text on a dashed line represents the join condition, in this case in the simplified form for equality of a shared attribute.

$v \in V$. For the meta-level transformation, we use the generation function Γ_M, while for the model-level transformation, we use the transformation function $\gamma_{M,V}$, as shown in (2). In practice, $\gamma_{M,V}$ is given by our transformation of the query to TGG rules and the subsequent execution of the TGG rules.

We illustrate the described operators by applying them to the example meta-models *Transport* and *Stops* shown in Figure 1. Note that the following examples are only meant to illustrate the operators and do not belong to a common view type definition and may, therefore, disagree with each other.

A. Selection and Join

The selection operator σ selects classes and relations between them from the source and maps them to the target. In the base case, we select a single class to be included in the target and specify the name of the target class. With our example meta-model *Transport*, shown in Figure 1, we could, e.g., write $\sigma(Line \Rightarrow TramLine)$. The target meta-model would then include a meta-class *TramLine*, and the model-level transformation would transform instances of *Line* to instances of *TramLine*.

In the more general case, we can specify patterns for the source and target selection. A pattern is a rooted directed graph, i.e., a directed graph with a single root, in which all other nodes are reachable from the root node. The nodes in a pattern are classes, while the edges are relations between classes. A class can be part of a selection pattern in at most one query. In this work, we limit patterns to chains, i.e., a node can have at most one incoming and outgoing edge.

The relations between the classes can be explicit or implicit. Explicit relations are given by a named reference with multiplicity 1 between two classes, e.g., $Line \rightarrow_{terminal} Station$. The name, *terminal* in this example, refers to a reference of the meta-class on the left-hand side of the reference operator $\rightarrow$. On the other hand, implicit relations are given by a join between two classes, relating their attributes with a join condition, e.g., $Station \bowtie_{Station.id = Stop.id} Stop$. In the simple case of a common attribute, resembling a natural join, we also write $Station \bowtie_{id} Stop$. In general, the join condition can be any boolean function on the attributes of the joined classes. Note that we again assume that exactly two instances are related by a join.

$$\sigma(Line \rightarrow_{terminal} Station \bowtie_{id} Stop \Rightarrow TramLine \rightarrow_{end} TerminalStop) \tag{3}$$

References and joins can be combined in a single pattern, and patterns can appear both on the source and target side of a selection operator. An example, using the meta-models *Transport* and *Stops*, shown in Figure 1, could be the query in (3), in which the classes *Station* and *Stop* are combined. An illustration of the source and target patterns is shown in Figure 2.

B. Filter

In addition to selecting classes and their relations, we can also filter the instances of the selected classes. For that, we employ the filter operator ϕ, which takes a predicate, i.e., a boolean function, on the attributes of all classes in the source selection pattern. The instances for which the predicate evaluates to true are transformed to the target. For our example meta-model, shown in Figure 1, we could write the query in (4), which selects all *Line* instances with a name starting with the character "S".

$$\sigma(Line \Rightarrow Line) \cdot \phi(startsWith(Line.name, \text{“S”})) \tag{4}$$

C. Attribute Projection

With the introduced operators, we can select classes and their relations, and restrict which instances we want to transform. To map the attributes of the selected classes and their instances to the target, we introduce an additional attribute projection operator π, which has two forms. With its first form $\pi(a \Rightarrow b)$, we can select an attribute a of one of the classes in the source selection pattern and specify its name and position b in the target selection pattern. Its more general second form $\pi(b := f(\ldots))$ can be used to calculate the value of an attribute b in the target selection pattern using a function f on the attribute values of all classes in the source selection pattern. A query on our example meta-model *Transit*, shown in Figure 1, utilizing both forms, is shown in (5).

$$\sigma(Line \Rightarrow TramLine) \cdot \pi(Line.name \Rightarrow TramLine.id) \cdot \pi(TramLine.displayName := concat(\text{“Tram Line ”}, Line.name)) \tag{5}$$

In the example, we select the class *Line* and rename it to *TramLine*. We then select its attribute *name*, rename it to *id*, and place it in the *TramLine* class in the target. In addition, we define a new attribute *displayName* in the target class

TramLine and calculate its value by prefixing the *name* of the *Line* instance with the text "Tram Line".

D. Reference Projection

The selection patterns, introduced in Section III-A, only account for a static number of relations between classes, e.g., references with multiplicity 1. To map arbitrary references between classes, we, in addition, define the reference projection operator ρ. We use this operator to select references from a selected class in the source model and map them to references of a target class. To specify the referencing and referenced class in the source and target, we use reference patterns. A reference pattern consists of two selection patterns connected by a single reference with multiplicity $*$. A simple example is shown in query (6), using the example meta-model *Transit*, shown in Figure 1.

$$\begin{aligned}&\sigma(\mathit{TransitGroup} \Rightarrow \mathit{Group}) \cdot \\ &\rho(\mathit{TransitGroup} \rightarrow_{\mathit{stations}} \mathit{Station} \Rightarrow \\ &\quad \mathit{Group} \rightarrow_{\mathit{stops}} \mathit{Stop})\end{aligned} \tag{6}$$

In the example, the reference *stations* of the source class *TransitGroup* is mapped to the reference *stops* of the corresponding target class *Group*. Note that we require that the source class *Station* is mapped to the target class *Stop* in a separate query.

As stated, the referencing class and referenced class can be part of a selection pattern, as long as there is only one reference that is mapped to the target. This is demonstrated in query (7), where we map the source reference *platforms* to the target reference *terminalExits*. We select the reference by starting from the source class *Line*, following the reference *terminal*, and joining the classes *Station* and *Stop*. This is a valid selection pattern, as all of them are 1-to-1 relations between classes. To select the target class of the reference, we follow the source reference *exit* from *Platform* to *Exit*. Again, this is a 1-to-1 relation and a valid selection pattern. Note that we, in general, require that the selected target class of the reference, in this case *Exit*, is mapped by another query. An illustration of the source and target reference patterns is given in Figure 3.

$$\begin{aligned}&\sigma(\mathit{Line} \Rightarrow \mathit{Line}) \cdot \\ &\rho(\mathit{Line} \rightarrow_{\mathit{terminal}} \mathit{Station} \bowtie_{\mathit{id}} \mathit{Stop} \\ &\quad \rightarrow_{\mathbf{platforms}} \\ &\quad \mathit{Platform} \rightarrow_{\mathit{exit}} \mathit{Exit} \\ &\quad \Rightarrow \\ &\quad \mathit{Line} \rightarrow_{\mathbf{terminalExits}} \mathit{Exit})\end{aligned} \tag{7}$$

A special case of references that can be transformed are containment references. While they can be selected and mapped as regular references, using the containment operator ρ, we additionally want to ensure that contained instances are only transformed when their containing instance is transformed as well. This way, we cannot have classes with an incoming containment reference that is not present for their instances.

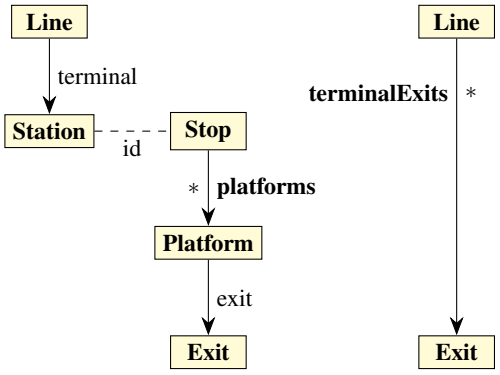

Fig. 3. Illustration of the source (left) and target (right) patterns for the example in (7). The dashed line represents a join between two classes. The text on a dashed line represents the join condition, in this case in the simplified form for equality of a shared attribute. References with multiplicity $*$ in a source meta-model are marked with a centered asterisk.

More generally, we may want to transform a non-containment reference in a source model to a containment reference in the target model, enforcing that each transformed instance of the referenced target class is contained. For that purpose, we introduce the variant ρ_c of the reference operator. As shown in the example in (8), it can be used in the same way as the standard variant.

$$\begin{aligned}&\sigma(\mathit{TransitGroup} \Rightarrow \mathit{Group}) \cdot \\ &\phi(\mathit{startsWith}(\mathit{TransitGroup.name}, \text{“K”})) \cdot \\ &\rho_c(\mathit{TransitGroup} \rightarrow_{\mathit{lines}} \mathit{Line} \Rightarrow \\ &\quad \mathit{Group} \rightarrow_{\mathit{lines}} \mathit{Line})\end{aligned} \tag{8}$$

In this example, we select the source class *TransitGroup* and transform its instances when their name starts with the character "K". We also select the containment reference *lines* to the class *Line*, which we select in another query. As we use the containment reference operator ρ_c, instances of the class *Line* are only transformed when they belong to a *TransitGroup* that is also transformed.

IV. Transformation to Triple Graph Grammar

In Section III we introduced the relational operators we construct queries of. As we motivated in Section I, we want to use these queries to generate views on models and to bidirectionally transform changes between them. To that end, we transform the queries to a triple graph grammar (TGG). Each query will result in exactly one triple graph rule, which combined make the TGG. The resulting TGG can be operationalized to transform models to views, views to models, and to synchronize existing models and views. In the following sections, we describe how the different operators can be expressed in triple graph rules.

A. Selection Patterns

The select operator σ, introduced in Section III-A, is used to select classes and relations in the source and to map them to the target. In a triple graph rule, this is achieved by adding the classes and references as green nodes and connecting them

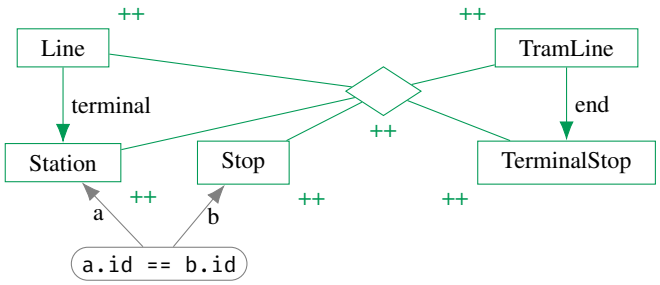

Fig. 4. Triple graph rule generated from the query in (3). Rounded rectangles represent attribute conditions, while diamond shapes represent correspondence nodes.

with a single correspondence node. The source classes and references are placed in the source part of the triple graph rule, while the target classes and references are placed in the target part of the triple graph rule. If during a transformation an instance of the source pattern is found, it is transformed to an instance of the target pattern and vice versa.

While references can be expressed in triple graph rules, our selection patterns also support joins as relations between classes. TGGs do not support joins directly, but as we limited our definition of joins to 1-to-1 relations between classes, we can emulate a join. We do this by adding an attribute condition to the triple graph rule, which constrains the application of the rule to instances of the classes that we want to join. In other words, the rule is only applied to instances that fulfill our join condition.

In Figure 4, we show the triple graph rule generated from the example query shown in (3), which is visualized in Figure 2. The three classes *Line*, *Station*, and *Terminal* in the source pattern appear as green nodes in the source part, as well as the reference *terminal*. The join condition `id` is transformed to the attribute condition `a.id == b.id`, where `a` refers to the node *Station* and `b` refers to the node *Stop*. Similarly, the classes *TramLine* and *TerminalStop*, as well as the reference *end* between them, appear as green nodes and edges in the target part. A single correspondence node connects all source and target nodes.

B. Filter

In addition to selecting classes and relations, we introduced the filter operator ϕ in Section III-B, which filters the transformed instances of classes in the source pattern. For transforming instances with a triple graph rule, this means we want to apply the triple graph rule only if the candidate instances fulfill the filter predicate. We can achieve this by adding an attribute condition, similar to the way we implemented the join condition.

Figure 5 shows an example of a triple graph rule, in which we apply a filter condition to the source class *Line*. Instance of the class are only transformed to instances of the target class *Line* when their name starts with the letter "S".

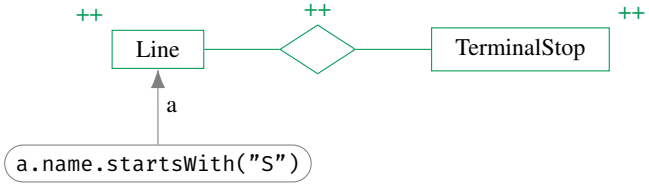

Fig. 5. Triple graph rule generated from the query in (4).

C. Attribute Projection

To map attributes selected with the attribute projection operator π, introduced in Section III-C, we again use attribute conditions. For this use case, however, we use attribute conditions between source and target nodes. To support model-to-view as well as view-to-model transformations, we therefore require attribute calculation functions for both directions. We realize this by providing a backward function for every attribute calculation function, as defined in (9).

$$\begin{aligned} f &: T_1 \to \cdots \to T_n \to U \\ f^{\leftarrow} &: U \to (T_1 \times \cdots \times T_n) \cup \bot \end{aligned} \tag{9}$$

We use the value $\bot$ to indicate that the target values cannot or should not be transformed to the source, and, consequently, the changes to the target should be rejected. Note that we do not require the backward function $f^{\leftarrow}$ to be the inverse function f^{-1} of f.

In the simplest case, realized by the first form $\pi(a \Rightarrow b)$ of the attribute projection operator, we apply the identity function $f(x) = x$ to transform the source value to the target. We can therefore always provide a backward function by using the inverse, which is the identity function itself, i.e., $f^{\leftarrow}(y) = y$. We apply this to realize the first instance $\pi(\mathit{Line.name} \Rightarrow \mathit{TramLine.id})$ of the attribute projection operator in the example in (5). The second instance of the attribute projection operator demonstrates its second form $\pi(b := f(a))$ and the more general case. In this example, we have the forward function $f(\mathit{name}) := \mathit{concat}(\text{"Tram Line "}, \mathit{name})$ and can provide a partial backward function. If the target value *id* has the prefix "Tram Line ", we remove the prefix; otherwise, we return $\bot$. Note that by *partial backward function*, we mean a total function that returns $\bot$ in some cases.

The triple graph rule for the example in (5) is shown in Figure 6. The two attribute projections are realized as attribute conditions between the involved source class *Line* and target class *TramLine*.

D. Reference Projection

In addition to mapping classes and their attributes, we introduced the reference projection operator ρ in Section III-D. In contrast to selection patterns, described in Section III-A, the reference projection operator maps references with multiplicity $*$. To achieve this in TGGs, the references have to be transformed separately from the classes. For that, we generate an additional triple graph rule in which we add the reference patterns, except the transformed reference, as context nodes

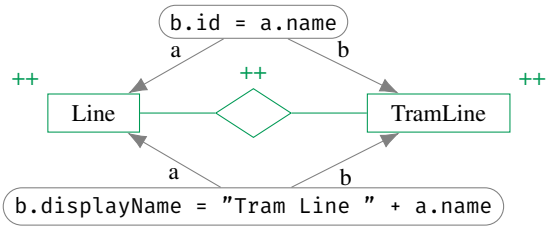

Fig. 6. Triple graph rule generated from the query in (5). For illustration purposes, we use the equality operator for the attribute conditions, which are implemented as forward and backward attribute calculation functions, as described in Section IV-C.

and edges to the triple graph rule. The transformed reference is added as a green reference between the referencing and referenced parts of the reference patterns in the source and target graph.

In order to match the right instances of the classes selected in the query, we have to include the selection patterns as context in the triple graph rule as well, including potential filter conditions. We require that the last nodes of the reference patterns be transformed in another query and add a correspondence node between them. The part of the reference patterns that is not covered by the selection patterns has to be matched elsewhere as well in order for us to include it as context nodes. These nodes can either be selected by other queries, or we generate additional triple graph rules matching them without transforming them to the view.

Figure 7 shows the triple graph rule generated from the example in (7), which is illustrated in Figure 3. The two parts of the source reference pattern, i.e., $Line \rightarrow_{terminal} Station \bowtie_{id} Stop$ and $Platform \rightarrow_{exit} Exit$, are added as context in the source graph, including the attribute condition realizing the join between the classes *Station* and *Stop*. We do the same for the target reference pattern, which, in this case, results in the single context nodes *Line* and *Exit*. As the selection patterns of our query, *Line* for the source and *Line* for the target, are already included in the reference patterns, we only add the correspondence node between them. In addition, we add a correspondence node between the last classes of the reference patterns, both called *Exit* and transformed in another triple graph rule. Finally, we add the green references *platforms* and *terminalExits* between the two parts of the reference patterns in the source and target graph, respectively.

E. Containment Reference Projection

While the definition of the containment reference operator ρ_c is quite similar to the definition of the reference operator ρ, both introduced in Section III-D, they have to be treated differently when transforming them to TGGs. The reason for that is that we only want to transform instances of contained classes when their respective container is transformed. In a triple graph rule, this can be expressed by adding the containing class as a context node for the transformation of the contained class. This does, however, mean that we have to consider the containment reference operator not when transforming

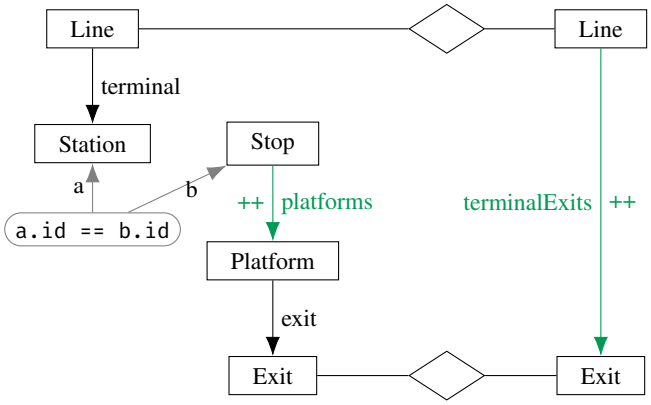

Fig. 7. Triple graph rule generated from the query in (7).

the containing class, but the contained class. As an example, in the query $\sigma(A \Rightarrow A) \cdot \rho_c(A \rightarrow_b B \Rightarrow A \rightarrow_b B)$, we map the containment reference b of the selected class A, but we actually have to consider the containment reference when transforming the class B.

To resolve this at the query level, we introduce an additional context operator κ, which describes an incoming containment reference. As in the transformation of reference operators, described in Section IV-D, we add the entire selection patterns of the query to the reference patterns in the context operator. An instance of the context operator is then added to all queries that transform the contained class B. Note that a query can have at most one instance of a context operator.

The example from above would then be resolved to the query $\sigma(A \Rightarrow A)$ and an addition to the query transforming B, e.g., $\kappa(A \rightarrow_b B \Rightarrow A \rightarrow_b B) \cdot \sigma(B \Rightarrow B)$. For the generation of the triple graph rules, we then assume that our queries do not contain instances of the containment reference operator ρ_c.

Applying this strategy to the example shown in (8), we first resolve the query transforming *TransitGroup*, resulting in the queries in (10). Note that for the example in (8), we assumed to have a query $\sigma(Line \Rightarrow Line)$ as well. After resolving, the second query in (10) contains an instance of the context operator κ, which specifies the containing class *TransitGroup* in the source and *Group* in the target, as well as the containment reference *lines*.

$$\begin{aligned} &\sigma(TransitGroup \Rightarrow Group) \cdot \\ &\phi(startsWith(TransitGroup.name, \text{“K”})) \end{aligned}$$

$$\begin{aligned} &\kappa(TransitGroup \rightarrow_{lines} Line \Rightarrow \\ &\quad Group \rightarrow_{lines} Line) \cdot \\ &\sigma(Line \Rightarrow Line) \end{aligned} \tag{10}$$

Transforming the second query in (10) produces the triple graph rule shown in Figure 8. For the selection operator σ, the class *Line* is added as a green node to the source and target graph, together with a correspondence node connecting them. In addition, the containing classes *TransitGroup* and *Group* from the context operator κ are added as context nodes to the source

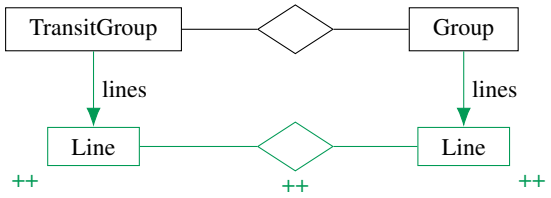

Fig. 8. Triple graph rule generated from the query in (10).

and target graph, respectively, also with a correspondence node between them. The containment reference *lines* between the containing and contained classes is added as a green reference, ensuring that an instance of a *Line* class is only transformed when it is contained.

V. Evaluation

To evaluate our approach, we implemented the operator model and transformation to triple graph grammars (TGGs) in a Java-based prototype. We use eMoflon::Neo [18] for the execution of our triple graph rules and their eMSL specification language as the output format of our transformation. Our prototype is included in the replication package [10] for this work.

As an initial evaluation of our approach, we consider the *applicability* of our operator model, the *correctness* of our transformation to TGGs, and the *scalability* of our combined approach. Since our operator model is intended to serve as an intermediate language for future view definition languages, as stated in Section I, we do not evaluate its usability. We evaluate our approach on two cases: an extension of our example meta-models shown in Figure 1, called *Transit* and included in the replication package, and a slightly adapted version of the movie library meta-model and view type from Burger [3], which we refer to as *Library*. Changes to the movie library meta-model were necessary to use it with the eMSL specification language [18]. For the *Transit* meta-model, we defined the three view types *Transit 1*, *Transit 2*, and *Transit 3*.

In addition to the current limitations of our approach, as presented in Section III and Section IV, our prototype currently only supports single classes in reference patterns. Thus, the query in (6) is supported, while the query in (7), transformed to the triple graph rule in Figure 7, is currently not supported. As demonstrated, this is not a limitation of our concept but rather a technical limitation of the current version of our prototype.

Applicability: We found that our operator model is applicable to the selected cases *Transit* and *Library*. With the changes required to use the eMSL specification language [18], we were able to write queries for all considered view types.

Correctness: To evaluate the correctness of our transformation from our operator model to TGGs, we executed the generated TGGs with eMoflon::Neo [18] and checked the results against a gold standard. We present the results of this evaluation in Table II. The model-to-view transformation, executed as a *forward* transformation in eMoflon::Neo, produced correct results for all cases. We used the *model integration* mode of eMoflon::Neo for the view-to-model transformation, assuming there is a model state for each view, from which it is derived. Executing the model integration for a changed view produced mostly correct results as well, with two issues, which we detail below.

TABLE II
Correctness of model-to-view (*Forward*) and view-to-model (*Model Integration*) transformations for different view types.

View Type	Forward	Model Integration
Transit 1	✓	✓
Transit 2	✓	✓[1]
Transit 3	✓	✓[1]
Library	✓	✓[1,2]

The first issue, indicated by the symbol ✓[1] in Table II, occurred in the model integration step for the view types *Transit 2*, *Transit 3*, and *Library*. It is caused by triple graph rules relating multiple source nodes on attributes, which are not transformed to the target, as, e.g., shown in Figure 4. For these rules, the model integration step failed, as incompatible attribute values were generated, causing the attribute condition to fail. To resolve this for arbitrary join conditions would require the developer to specify how to generate the values of all attributes of green nodes referenced in the join condition that are not transformed back from the view.

The second issue, indicated by the symbol ✓[2] in Table II, only occurred in the model integration step for the view type *Library*, but could occur in forward transformations as well. It is caused by conflicting triple graph rules, in this case for superclasses and subclasses. When a subclass is transformed by eMoflon::Neo, the rule for transforming its superclass can be executed instead, as it also matches, which was the case in our transformation. To resolve this would require dependencies or priorities for triple graph rules to control their application in conflict situations.

Scalability: For the scalability evaluation of our approach, we applied the same cases as for the correctness evaluation. As the size of the source meta-models does not necessarily relate to the size of the view type, e.g., when selecting only a subset of its elements, we compare the size of the view type to the number of queries required and the number of triple graph rules generated. We measure the size of a meta-model by counting its elements. The results of our evaluation are shown in Table III.

The number of queries required for the definition of a view type is at most equal to the size of the view type, as can be seen in Table III. This is because each query must contain a selection operator, as described in Section III, which maps at least one element from a source to the target. The number of queries does not have to be equal to the size of the view type, as can be seen for the view type *Library*, as a selection operator can map multiple source elements in a single query.

We further see that from a single query, multiple triple graph rules can be generated, as is the case for the view types

TABLE III
SCALABILITY OF VIEW DEFINITIONS BY COMPARING THE NUMBER OF META-MODEL ELEMENTS IN A VIEW TYPE (*VT Size*) TO THE NUMBER OF QUERIES (*# Queries*), THE NUMBER OF REFERENCE PROJECTIONS IN ALL QUERIES (*# References*), AND THE NUMBER OF GENERATED TRIPLE GRAPH RULES (*# Rules*) FOR DIFFERENT VIEW TYPES.

View Type	VT Size	# Queries	# References	# Rules
Transit 1	3	3	0	3
Transit 2	4	**4**	**1**	**5**
Transit 3	2	2	0	2
Library	**6**	**5**	4	9

Transit 2 and *Library* in our evaluation. The reason for this is that reference projections in a query generate additional rules for the transformation of the references, as described in Section IV-D. Analyzing the transformation from our operator model to TGGs, we find that for queries without reference projections, we generate exactly one rule, while we generate one additional rule per reference projection. The number of rules is therefore $n+k$ where n is the number of queries and k is the number of reference projections in all queries. Our reasoning is supported by the data we collected from our cases, shown in Table III.

VI. Related Work

The view definition language *ModelJoin* [4] defines an extensive set of relational operators for model queries. In addition to the operators we define in Section III, they support different join types, aggregation, and inheritance references. Due to this, we are not yet able to express all ModelJoin queries in NeoJoin. From their operators, they generate QVT-Operational [14] rules for the model-to-view and view-to-model transformations. In contrast, we transform our operators to triple graph grammars (TGGs), which, in addition, enables the incremental execution of transformations.

Greenyer and Kindler [7] implement a transformation from QVT-Core [14] to TGGs and discuss an approach to transform QVT-Relations [14] to TGGs. Similar to our work, they transform a relational language to TGGs; however, we argue that relational operators are more abstract and thus enable a more flexible view definition process. In addition to the transformation to TGGs, they compare the features of QVT and TGGs and propose several extensions to TGGs. One of these extensions is the introduction of reusable nodes, lifting the restriction of TGGs that every node in the graph must be bound exactly once. This restriction has led to issues in the transformation of reference operators to TGGs in our work as well, described in Section IV-D.

VII. Future Work

In this work, we introduced our relational operator model for the specification of view types and view generation, as well as the transformation from our operators to triple graph grammars (TGGs). As both are yet in an early stage of research, there are a number of open research questions and ideas, which we discuss in this section.

Selection: When introducing selection patterns in Section III-A, we defined a pattern as a rooted directed graph, i.e., a directed graph with a single root, from which all other nodes are reachable. After that, we limited patterns to chains of classes, connected by references or joins, which makes the subsequent definitions easier and is a limitation of the current version of our prototype. As this is not a conceptual issue, we plan to lift this restriction in the future. A more severe, and conceptual, limitation is the restriction of joins to 1-to-1 relations between instances in Section III-A. The reason for this is further detailed in Section IV-A and stems from the inherent limitation of TGGs that each node in a graph is matched exactly once [7]. To our knowledge, it is therefore not possible to create full cross products with TGGs, which would be the basis for more expressive joins. In the future, we want to research whether a pre-processing step, generating a full cross product of references, and triple graph rules applying the join conditions on them would be sufficient to express arbitrary joins in our operator model.

References: One of the larger limitations of our operator model is the restriction of the reference operator ρ to map only single references with multiplicity $*$, as described in Section III-D. Lifting this restriction is not straightforward, as a combination of references would have a direct impact on the number and structure of the generated triple graph rules. As an example, mapping two subsequent references, both with multiplicity $*$, would require two separate TGG rules, one for transforming instances of the first reference and one for transforming the instances of the second reference. In the future, we plan to collect specific patterns of references, their transformation to triple graph rules, and research their ability to specify bidirectional transformations. Another limitation, although not as severe, of the containment reference operator ρ_c is the restriction that a class, on the meta-level, can only be contained by a single other class. This is actually a limitation of the resolving step from instances of the containment reference operator ρ_c to instances of the context operator κ, described in Section IV-E. As having multiple containment references to a single class would require a disjunctive combination of contexts for that class, we would have to duplicate the affected queries and add different instances of the context operator κ to them. We plan to implement this extension to the resolving step and research its implication on the expressiveness of our operator model, as well as on the scalability of the transformation, discussed for the current version of the prototype in Section V.

Aggregation: One feature present in relational algebra, which we have not included in our operator model yet, is the grouping of multiple instances and the computation of attribute values with an aggregation function. Similar to cross-product joins, this is an inherent limitation of TGGs, as transforming an arbitrary number of nodes to a single aggregated node would require updating an existing node, which is not possible. While multi-amalgamation [13] could be a solution to this by combining a kernel rule, matched once, with multi-rules that can be matched multiple times, we are not aware of existing research on the combination

of multi-amalgamation and attribute conditions. Instead, we therefore plan to implement a fold operation, as in functional programming languages, aggregating values from an arbitrary number of instance nodes in the source graph step-by-step. The intermediate results of the fold steps would be stored in a chain of nodes in the target graph, resulting in one target node per source node. In a post-processing step, we would then remove the intermediate nodes and retain only the final, aggregated value of the last node in the chain in the target graph.

Prototype and Evaluation: As discussed in Section V, there are still open issues with the current version of our prototype, which we plan to investigate. In addition, we plan to extend the evaluation to larger and more realistic cases, such as the smart grid case from Burger, Mittelbach, and Koziolek [4].

VIII. Conclusion

In this paper, we presented an operator model for relational queries on heterogeneous models, supporting the selection, filtering, and projection of classes, their attributes, and references. We discussed their expressiveness and gave examples of queries on a small meta-model. In addition, we presented a transformation of queries to triple graph grammars, using attribute conditions for joining and filtering instances, as well as for projecting attributes. Building on the example queries, we presented triple graph rules to illustrate the transformation. To evaluate the applicability of our operator model, the correctness of our transformation, and the scalability of their combination, we presented a Java-based prototype implementation. We performed an initial evaluation on two cases, including meta-models, models, and view definitions. Finally, we discussed future research questions to improve the expressiveness of our selection and reference operators, as well as to add support for aggregations.

Although our work is in an early stage, our results show that our operator model could serve as an intermediate language for future view definition languages. As such, our approach enables view definition languages fulfilling our initial requirements, stated in Section I. As an application of our operator model and transformation, we plan to integrate them with the view definition language *NeoJoin* [11], proposed by König et al. We further plan to extend our evaluation to more realistic cases, such as the smart grid case of Burger, Mittelbach, and Koziolek [4]. In the end, supporting the flexible definition of view types and view generation transformations through relational queries on heterogeneous models could enable developers to create representations tailored to their tasks and support the consistent development of complex systems.

References

[1] Colin Atkinson, Christian Tunjic, and Torben Moller. "Fundamental Realization Strategies for Multi-view Specification Environments". In: *2015 IEEE 19th International Enterprise Distributed Object Computing Conference*. Adelaide, Australia: IEEE, Sept. 2015, pp. 40–49. ISBN: 978-1-4673-9203-7. DOI: 10.1109/EDOC.2015.17.

[2] Hugo Bruneliere et al. "A feature-based survey of model view approaches". In: *Software & Systems Modeling* 18.3 (June 2019), pp. 1931–1952. ISSN: 1619-1374. DOI: 10.1007/s10270-017-0622-9.

[3] Erik Burger. "Flexible Views for View-based Model-driven Development". PhD. Karlsruhe: KIT, 2014. URL: https://publikationen.bibliothek.kit.edu/1000043437.

[4] Erik Burger, Victoria Mittelbach, and Anne Koziolek. "View-based and Model-driven Outage Management for the Smart Grid". In: *11th International Workshop on Models@run.time (MRT) co-located with MODELS*. Ed. by S. Götz. Vol. 1742. CEUR Workshop Proceedings. Saint Malo, France: RWTH Aachen, Oct. 2016, pp. 1–8. URL: https://st.inf.tu-dresden.de/MRT16/papers/MRT16_paper_1.pdf.

[5] E. F. Codd. "A relational model of data for large shared data banks". In: *Communications of the ACM* 13.6 (June 1970). Publisher: ACM, pp. 377–387. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/362384.362685.

[6] Istvan David, Hans Vangheluwe, and Eugene Syriani. "Model consistency as a heuristic for eventual correctness". In: *Journal of Computer Languages* 76 (July 2023). Publisher: Elsevier, p. 101223. ISSN: 25901184. DOI: 10.1016/j.cola.2023.101223.

[7] Joel Greenyer and Ekkart Kindler. "Comparing relational model transformation technologies: implementing Query/View/Transformation with Triple Graph Grammars". In: *Software & Systems Modeling* 9.1 (Jan. 2010), pp. 21–46. ISSN: 1619-1374. DOI: 10.1007/s10270-009-0121-8.

[8] ISO. *42010 — Software, Systems and Enterprise — Architecture Description*. Nov. 2022. URL: https://www.iso.org/standard/74393.html (visited on 07/23/2024).

[9] Heiko Klare et al. "Enabling consistency in view-based system development – the Vitruvius approach". In: *Journal of Systems and Software* 171 (2021). Publisher: Elsevier, p. 110815. DOI: 10.1016/j.jss.2020.110815.

[10] Lars König, Daniel Ritz, and Erik Burger. *Replication Package for "Transforming Relational Model Queries to Triple Graph Grammars"*. July 2025. DOI: 10.5281/ZENODO.15849163.

[11] Lars König et al. *Towards dynamic views on heterogeneous models – the NeoJoin view definition language*. Preprint, to appear in the Joint Proceedings of the STAF 2025 Workshops. 2025. DOI: 10.5445/IR/1000182351.

[12] Leen Lambers et al. "Attribute Handling for Bidirectional Model Transformations: The Triple Graph Grammar Case". In: *Bidirectional Transformations 2012*. Vol. 49. Electronic Communications of the EASST. July 2012. DOI: 10.14279/tuj.eceasst.49.706.

[13] Erhan Leblebici et al. "Multi-amalgamated Triple Graph Grammars". In: *Graph Transformation*. Ed. by Francesco Parisi-Presicce and Bernhard Westfechtel. Cham: Springer International Publishing, 2015, pp. 87–

103. ISBN: 978-3-319-21145-9. DOI: 10.1007/978-3-319-21145-9_6.

[14] *MOF Query/View/Transformation (QVT) Specification*. June 2016. URL: https://www.omg.org/spec/QVT/1.3/.

[15] Douglas C. Schmidt. "Guest Editor's Introduction: Model-Driven Engineering". In: *Computer Society* 39.2 (Feb. 2006), pp. 25–31. ISSN: 0018-9162. DOI: 10.1109/MC.2006.58.

[16] Andy Schürr. "Specification of graph translators with triple graph grammars". In: *Graph-Theoretic Concepts in Computer Science*. Ed. by Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer. Berlin, Heidelberg: Springer, June 1994, pp. 151–163. ISBN: 978-3-540-49183-5. DOI: 10.1007/3-540-59071-4_45.

[17] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database system concepts*. 6th ed. New York: McGraw-Hill, 2011. ISBN: 978-0-07-352332-3 978-0-07-128959-7.

[18] Nils Weidmann and Anthony Anjorin. "eMoflon::Neo – consistency and model management with graph databases." In: *STAF workshops*. 2021, pp. 54–64. URL: https://bx-community.wdfiles.com/local--files/bx2021:program/WA21.pdf.

[19] Nils Weidmann, Robin Oppermann, and Patrick Robrecht. "A feature-based classification of triple graph grammar variants". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2019. New York, NY, USA: ACM, Oct. 2019, pp. 1–14. ISBN: 978-1-4503-6981-7. DOI: 10.1145/3357766.3359529.