Efficient Model Synchronization with View Triple Graph Grammars

Anthony Anjorin*, Sebastian Rose, Frederik Deckwerth, and Andy Schürr

Technische Universität Darmstadt
Real-Time Systems Lab.
Merckstr. 25
64283 Darmstadt, Germany
name.surname@es.tu-darmstadt.de

Abstract. Model synchronization is a crucial task in the context of Model Driven Engineering. Especially when creating and maintaining multiple suitable abstractions or *views* of a complex system, a *bidirectional transformation* is required to keep all views and the corresponding system synchronized by automatically propagating changes in both directions. Triple Graph Grammars (TGGs) are a declarative, rule-based bidirectional transformation language, which can be used to support model synchronization. In practice, most TGG tools restrict the supported class of TGGs for efficiency reasons. These restrictions are, however, seldom intuitive and are often difficult to understand and adhere to, especially for non-experts. *View* Triple Graph Grammars (VTGGs) are a restricted form of TGGs, which can be highly optimized for efficient view update propagation. We argue that the restrictions posed by VTGGs are explicit and intuitive for users, as they can be adequately motivated based on the main application scenarios for VTGGs. In this paper, we present for the first time a *formalization* of VTGGs, stating precisely the advantages and limitations of VTGGs as compared to TGGs, and backing our claims with initial runtime measurements from a practical case study.

Keywords: model driven engineering, bidirectional model transformation, triple graph grammars, view triple graph grammars.

1 Introduction and Motivation

It is usually impossible to invest the time required to gain a deep understanding of a *complete* system and, therefore, a crucial task is focusing on relevant aspects with task-specific *views*. Although views are crucial for productivity, maintaining a view *manually* is infeasible for most practical applications. Generated, read-only views are also unsatisfactory as an important requirement is being able to apply changes to the underlying system at the level of abstraction provided by

* The project on which this paper is based was funded by the German Federal Ministry of Education and Research, funding code 01IS12054. The authors are responsible for all contents.

J. Cabot and J. Rubin (Eds.): ECMFA 2014, LNCS 8569, pp. 1–17, 2014.

the view. In general, a *bidirectional transformation* is required to keep all views synchronized by automatically propagating *updates* in both directions.

In a *Model-Driven Engineering* (MDE) context, *Triple Graph Grammars* (TGGs) are a rule-based, formally founded bidirectional model transformation language [8], which can be used to keep two different but related models synchronized [8]. TGGs can be used to specify views, but supporting *lightweight*, i.e., simple but very efficient, view update propagation is challenging as no restrictions are made and arbitrary source and target (i.e., view) structures are allowed. In application scenarios where the target model is, however, clearly a *view* of the source, i.e., a true abstraction/simplification, certain restrictions apply naturally to the structure of the view and can be exploited to enable efficient view update propagation. Examples include (i) reducing the complexity of a transformation by applying suitable views, and (ii) using (a chain of) views to provide a common abstraction on different structures, so that the same transformations can be applied to them. In both cases, the subsequent transformations must be able to treat views exactly as normal models, leading naturally to a set of restrictions, e.g., that nodes in the view can be created in any order. In cases where these restrictions apply, *View Triple Graph Grammars* (VTGGs), first introduced by [7], can be used as a restricted form of TGGs to realize truly lightweight views of source structures, which are specially optimized for efficient update propagation.

In practice, all TGG tools we are aware of pose certain restrictions on the class of supported TGGs to guarantee efficient synchronization (depending on the size of changes and not on the size of models). These restrictions are, however, seldom intuitive for users, i.e., are typically technical and cannot be directly motivated from the supported application scenarios. In this paper, we (i) argue that the restrictions for VTGGs are intuitive for users as they are directly connected to their intended usage, (ii) present for the first time a formalization of VTGGs as a restricted form of TGGs, giving straightforward proofs for the expected formal properties according to [8]. Furthermore, we clarify the exact difference between TGGs and VTGGs with a qualitative and quantitative comparison.

Section 2 presents our running example, explaining intuitively how VTGGs can be used for view specification and synchronization. Section 3 compares our approach to related work, Sect. 4 formalizes VTGGs as a restricted form of TGGs, and Sect. 5 provides runtime measurements from our case study. Section 6 concludes with a summary and an overview of future work.

2 Running Example

Our running example is inspired by a real-world project in which a client-server application is developed using the Eclipse Rich Client Platform (RCP) as a basis for the client. An Eclipse RCP ecosystem consists of multiple *plugin projects* (folders in the filesystem) in a *workspace* (the root folder). Figure 1 depicts such a project (Reservation) to the left. Every project contains a source folder src with Java code and other resources (e.g., icons), and a plugin.xml file. All concepts described so far are supported by the standard Java *Package Browser*.

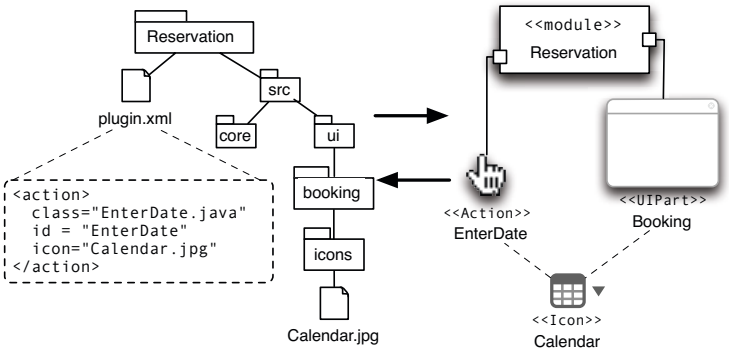


Fig. 1. Example for source and view models in the running example

Some high-level concepts used by developers are depicted to the right of Fig. 1: *Modules* represent business related containers such as the Reservation module, which provides all functionality related to the domain concept "reservation". A module consists of *UI parts*, i.e., aspects relevant for the user interface such as icons and dialogues. Modules also contain units of functionality referred to as *actions*, which can be related with an icon in the user interface, e.g., the EnterDate action, which is visualized with the Calendar icon in the UI part Booking. Note the depicted action fragment in the plugin.xml file and the folder structure, which correspond to the action and the UI part in the Reservation module, respectively. Representing projects using these high-level concepts not only supports communication amongst domain experts, but also provides a compact representation and enforces conventions that hold for all projects/modules.

2.1 Models, Metamodels, and Model Transformation

A *model* is a graph, i.e., a structure consisting of objects and links, with source and target functions connecting links to their source and target objects, respectively. Relationships between models are formalized as graph morphisms, which are structure preserving maps between graphs, i.e., maps that preserve how links are connected to objects. The conformance relationship between a model and its *metamodel* is a graph morphism "type" between one graph (the model) and another graph (the metamodel). Objects and links in metamodels are referred to as *classes* and *associations*, respectively. This algebraic formalization of MDE concepts can be extended to include further details such as type preserving morphisms, attributes (which are also typed), inheritance and abstract classes [6].

The metamodels for our running example are depicted in Fig. 2. The *source metamodel* on the left represents a simple tree structure, which can be extracted from an Eclipse project using standard parsers to produce trees from the folder structure and the different files. Concepts include, therefore, Folders, Files, and labelled Nodes with Attributes. On the right, the *view metamodel* represents the high-level concepts: Modules and Actions that reference Icons in a UIPart. The metamodel in the middle is referred to as the *correspondence metamodel* as it

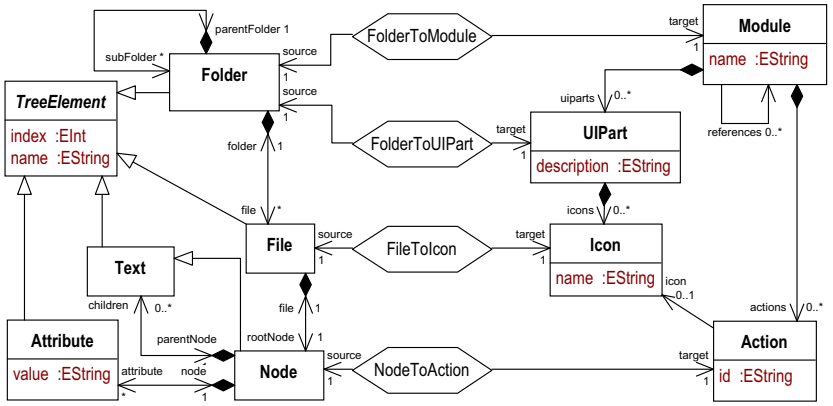


Fig. 2. Triple of metamodels for the running example

states which source and target elements correspond to each other. Note that correspondence types are depicted as hexagons only to improve readability.

A *model transformation* takes an input model M_I and produces an output model M_O. In a rule-based approach, this is formalized as *applying* a transformation *rule* r to the input model M_I. The rule $r = (L, R)$ consists of a precondition L and a postcondition R, which are both *patterns* that have to be mapped to actual model elements when applying the rule. The result of this mapping is referred to as a *match*. If a match $m(L) \subseteq M_I$ can be determined for the precondition L in the input model M_I, the rule r can be applied by replacing the precondition by the postcondition to yield the output model M_O.

2.2 View Specification with VTGG Rules

The formalization of rule-based model transformation mentioned above can be extended to *triples* of models, i.e., the precondition L and postcondition R of rules are triples of source, correspondence and target patterns. Such triple rules are referred to as *VTGG rules*, which specify a view by describing how the view model evolves *together* with related correspondence and source models.

The first step in specifying a view with a VTGG is to describe what it means to instantiate all concrete classes in the view metamodel, i.e., to create objects in the view. Such rules are referred to as *class rules* and exactly one class rule is required for every concrete (non-abstract) class in the view metamodel. Let us refer to this requirement as *Restriction 1*. Figure 3 depicts the class rules for the running example using a compact notation that combines L and R in the following manner: *Created* elements (green with a ++ stereotype) are created by the rule to fulfil the postcondition, i.e., are in $R \setminus L$, while *context* elements (black without any stereotype) constitute the precondition L. Regard the rule ModuleRule, which states that creating a module in the view corresponds to creating a root folder (folder) with an XML file (plugin, root) and a source folder structure (srcFolder, core, ui). Simple attribute conditions can be in-lined in nodes such

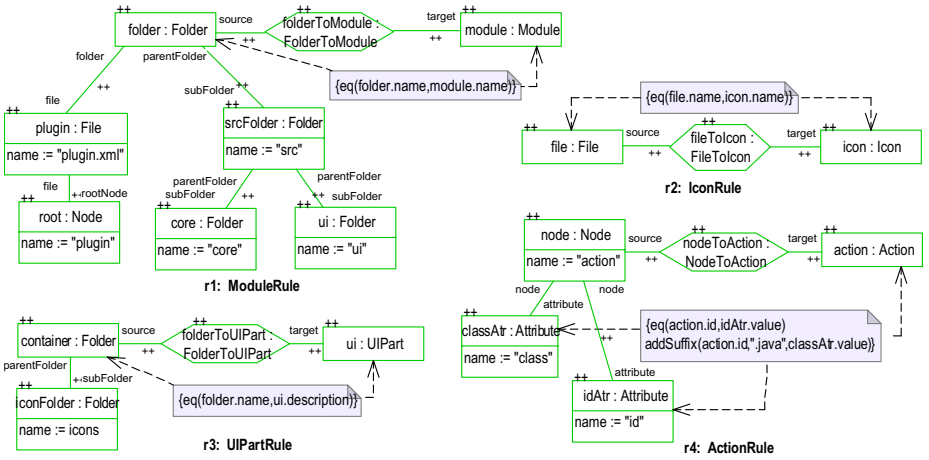


Fig. 3. VTGG class rules for the running example

as name:="plugin" in root, or specified with a separate, bidirectional constraint language such as for eq(folder.name, module.name), which can be extended to include complex user-defined constraints [1]. VTGG class rules only create elements without demanding any context, meaning that view objects can always be created in any order. Let us refer to this requirement as *Restriction 2*.

After defining how view objects can be created, the next step in the process is to describe what it means to *connect* existing view objects, i.e., to create *links* in the view by instantiating associations in the view metamodel. Such rules are referred to as *association rules* and exactly one association rule is required for every association in the view metamodel. This is an extension of *Restriction 1*.

Figure 4 depicts the association rules for the running example. In contrast to class rules, association rules contain context elements (black without any stereotype), which must have already been created with other rules before the association rule can be applied (precondition must be fulfilled). Association rules are only allowed to augment the view by creating a link between two existing view objects. For example, ModuleUIPartRule connects a Module with a UIPart. In the source model, association rules can create an arbitrary structure as long as the required context follows from the existence of the view objects. For ModuleUIPartRule, this means that the required folder structure (folder, srcFolder, uiFolder) and container folder are guaranteed to exist as they are created together with module and ui by the corresponding class rules. This is an extension of *Restriction 2*. The complete specification comprises two further association rules, r_7 and r_8, for the remaining two associations in the view metamodel.

As the view is typically an abstraction of the source, the final step is to specify in exactly what ways the source can evolve *without* affecting the view. These rules are referred to as *idle rules* and can either be *class idle* or *association idle*, depending on the required context. Figure 5 depicts two idle rules for the running example. IdleModuleRule is an idle class rule and allows the addition of

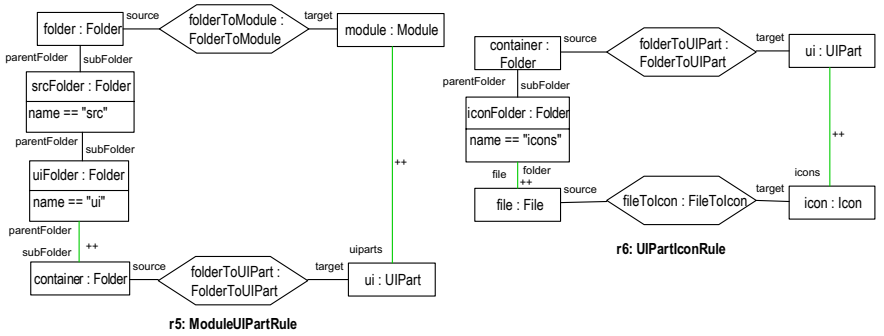


Fig. 4. VTGG association rules for the running example

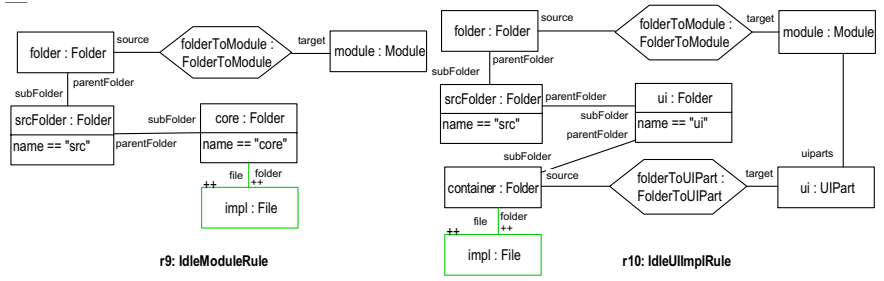
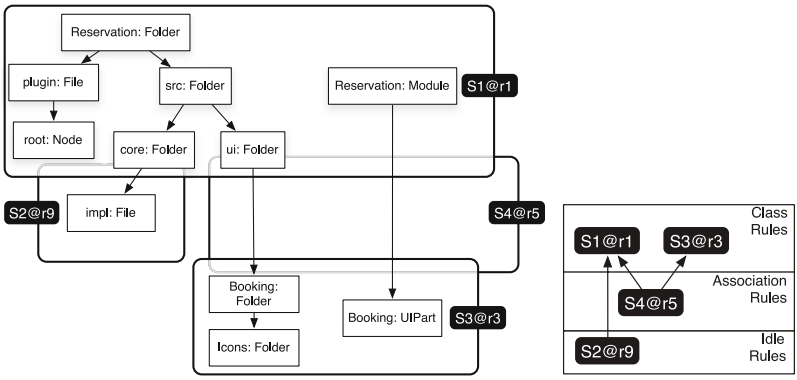


Fig. 5. VTGG idle rules for the running example

arbitrarily many implementation files to the core source folder without affecting the view in any way. Idle class rules can only require context provided by a class rule, implying that the created elements (impl in the case of IdleModuleRule) must be deleted together with the required view object (module). IdleUIImplRule is an idle association rule, which allows the addition of implementation files for a UIPart. Idle association rules can require context provided by an association rule, and in this way, IdleUIImpRule enforces that a UIPart can only be implemented if it is already part of a module. This has consequences that should be discussed with domain experts, e.g., the implementation of a UIPart must be deleted as soon as it is detached from its Module in the view to retain consistency.

2.3 Model Synchronization with VTGGs

To explain intuitively how VTGGs can be used for synchronization, Fig. 6(a) depicts a source model with a corresponding view model, both annotated with labels to denote VTGG rule applications. Fig. 6(b) depicts the corresponding rule application *dependency* graph, which is used to explain the synchronization algorithm. The following concrete scenario could have led to this triple: A user started with a source model consisting of a root folder Reservation with a plugin.xml file and src, src/core and src/ui subfolders. The user now decides to create



(a) Consistent Source and View Models (b) Dependency Graph

Fig. 6. Exemplary transformation sequence for the running example

a view model via a *forward transformation*, i.e., a source-to-view transformation, which parses the source structure and creates a Reservation module in the view. This first step (S1) is consistent with the application of the VTGG rule r1 in the simultaneous build-up and the result is labelled with S1@r1 in Fig. 6(a). The user now adds an implementation file to the core folder and updates the view by repeating the forward transformation. The source structure is parsed as S1@r1, S2@r9, meaning that the view remains unchanged (S2@r9 is an application of an idle class rule and does not change the view). The user now creates a UIPart in the view named Booking and adds it to the Reservation module. These view updates are reflected in the source model via a *backward transformation*, i.e., a view-to-source transformation, which creates the corresponding source structure resulting in a triple now consistent with the rule application sequence S1@r1, S2@r9, S3@r3, S4@r5. Finally, the user deletes the Reservation module from the view. This view update is propagated using the rule application dependency graph (Fig. 6(b)). To "reverse" S1@r1 (i.e., to delete the Reservation module), all dependent rule applications have to be recursively reversed first. This means that S2@r9 (all implementation files in the core folder), as well as S4@r5 (all incident links in the view) must be reversed first, resulting in a triple consistent with S3@r3. Due to the restricted structure of VTGG rules, the dependency graph always consists of three levels as depicted in Fig. 6(b). Changes in the view are thus guaranteed by construction to have a "local" influence (to only affect dependent rule applications on lower levels).

2.4 VTGG as Restricted TGGs

Based on our example, we can now list and discuss (for now informally, cf. Sect. 4.2 for the formalization) the main restrictions posed by VTGGs. Theoretically, TGGs can be used directly (without any restrictions) to specify views.

In practice, however, due to efficiency requirements of real-world applications, all TGG tools we are aware of impose some set of restrictions (typically related to *confluence* [6]) on the class of supported TGGs. These restrictions are typically technical and have nothing to do with a focussed application domain. With VTGGs this is different, as the set of restrictions are directly connected to the intended usage of VTGGs, namely establishing a *new* view metamodel. Instances of this view metamodel are to be indistinguishable from normal models, and are typically used as a suitable high-level interface for further transformations. This has two implications: (i) it must be possible to create single view objects in an arbitrary order, and (ii) view links can be created as soon as the view objects to be connected are present.

Remarkably, these core requirements coincide perfectly with the main VTGG restrictions we have explained with our running example so far: (i) there must be a class/association rule for every concrete class/association in the view metamodel (Restriction 1), and (ii) class rules do not require context elements, while association rules only demand context guaranteed by the class rules for the corresponding source and target classes of the association in the view. A consequence is that association rules cannot depend on other association rules (Restriction 2). Both restrictions are exploited to control ripple effects of update propagation, i.e., VTGG rule dependency graphs always consist of three levels with dependencies allowed only to lower levels. The corresponding limitation of VTGGs, compared to TGGs, is that every instance of the view metamodel can be generated by the VTGG, i.e., the user has no means of constraining the induced view language in any way. VTGGs are, therefore, not suitable when this is required.

A further optimization employed by VTGGs follows from the fact that the view language is being newly specified and that it is thus allowed to add extra technical references as required to the view metamodel. In stark contrast, TGGs aim to be non-invasive and can be used for existing and unchangeable metamodels. Being able to manipulate the view metamodel means that the extra correspondence model can be reduced to a set of references leading directly from view to source elements. This is a technical point and has no effect on the formalization but is still an important VTGG restriction/optimization with a substantial effect on efficiency. Finally, as there are no idle view rules, the view can always be created from the source. This indicates a clear focus on optimizing the backward propagation (propagating changes to the view), which is greatly simplified and can be efficiently implemented as the structure of all rules on the view side is kept very simple. Forward propagation can be realized with a standard TGG algorithm either in batch (the view is re-created completely from scratch) or incremental mode (the old view is updated). We do not aim to replace TGGs and an ideal combination would be to use VTGGs as a preprocessing step for TGGs, i.e., to retrieve high-level views of low-level source and target models, before using TGGs to synchronize the views, leading to clearer, more concise TGG rules. In general, although VTGGs limit expressiveness (the language of allowed view models cannot be restricted, e.g., the running example would not be a VTGG if the roles of view and source were swapped), they provide an

efficient means of simplifying (modularizing) TGG or standard graph transformation rules that can operate on these views exactly as if they were normal, stand-alone models.

3 Related Work

In this section, we classify related approaches into three main groups corresponding to the following questions:(1) How is our approach to VTGGs different from that of [7]? (2) What connections and parallels exist to the *Asymmetric Delta Lens* (ADL) framework? (3) How does the alternative definition of views in the algebraic graph transformation framework compare to ours?, and (4) What other view specification approaches, less related to VTGGs, exist?

(1) Materialized vs. Non-materialized Views: The basic idea of VTGGs for the specification of views is introduced in [7]. The *class adapter* pattern, employed by [7], alters the *source metamodel*, introducing inheritance relationships so that certain source elements can additionally take on the role of view elements. Although this enables efficient memory usage via *non-materialized views*, i.e., views that do not exist as separate models but are rather represented as an additional *role* taken on by certain elements in the source model, it complicates view composition as it is unclear how to successively adjust the inheritance relationships appropriately. In contrast to [7], our approach uses the *object adapter* pattern, maintaining our view as a lightweight wrapper for the source model. Our approach results in *materialized views*, i.e., a separate view model, and is a good compromise as the views can still be kept lightweight, e.g., by keeping all attribute values in the source model and computing attribute values for the view on demand. Furthermore, in contrast to [7], we provide a constructive formalization of VTGGs in the context of TGGs [8], which is useful as a basis for concrete VTGG implementations.

(2) Connections to the ADL Framework: Compared to (V)TGGs, the Asymmetric Data Lens (ADL) framework [4] is a more generic and abstract formalization of bidirectional transformations. The framework captures the intuitively expected behaviour of views with a set of *lens laws*. Due to its generality, it offers a formal foundation that can be used to draw parallels between very different approaches. This higher level of abstraction is, however, not only an advantage as it is impossible to formulate, e.g., a precise notion of efficiency such as in the TGG framework. For the reader versed in both frameworks, the following points give a brief overview of connections and differences:

1. The "sanity" laws from the ADL framework (incidence and identity preservation) are fulfilled for (V)TGGs by exploiting the correspondence model and do not have to be demanded explicitly.
2. Correctness for (V)TGGs corresponds to *PutGet* (correctness of backward propagation) in the ADL framework and is also used to fulfil the compositional laws. (V)TGG are, however, potentially more flexible (non-functional) and **get** is allowed to be a relation and not a function (there can be multiple consistent triples for the same source model).

3. Completeness for (V)TGGs is implicitly demanded in the ADL framework by requiring totality for **get** and **put** on the source and view model spaces.
4. Further (V)TGG properties such as efficiency or expressiveness are irrelevant in the abstract ADL setting.

(3) Views in the Algebraic Graph Transformation Framework: The algebraic graph transformation framework of [6] is extended and generalized in [5], defining typed graph morphisms between *different* type graphs and *different* data signatures, used to specify the direct connection between a view and its corresponding source. In contrast, our TGG-based formalization uses correspondence elements, i.e., a separate model, to formally specify the connection between view and source. A view itself is, therefore, not the direct result of a transformation rule, but is defined by the rules of the VTGG and the induced consistency relation. Although [5] provides the basis for a potentially simpler formalization of view specification frameworks, our TGG-based approach can make use of the substantial existing TGG formalization, and can be realized as an extension/adaptation of our existing TGG implementation.

(4) Other Approaches to View Specification: The need for views in an MDE context has led to an Eclipse project *EMF Facet*[1] that allows the specification of queries on models using Java or XPath. Compared to our rule-based, declarative VTGG approach, such queries in Java / XPath are often rather low-level and the derived EMF facet views are read-only. With a clear focus on modularity, views are used in [11] to hide details and to establish interfaces between modules. The main motivation is the modelling of large and distributed systems, where dependencies are to be reduced as much as possible. Although VTGGs can also be used for modularization and defining interfaces, our main motivation is rather to provide an efficient suitable abstraction for further manipulation with TGGs or normal graph transformations. Compared to [11], views with VTGGs are specified with an explicit set of declarative rules, while [11] uses a high-level mapping from which required transformations for the view are automatically derived. The latter is, consequently, on a more abstract level but is less expressive. In the domain of database engineering, specifying a view as a set of queries is a mature and established practice with a solid formal foundation [3]. In other domains, bidirectional programming languages [12], as well as lens implementations for strings [2] are used to support view specification. As a realization of the *viewpoint* framework presented in [10], Xlinkit [9] provides a declarative, rule-based means of specifying views on XML files. In an MDE context, however, where typed graph structures are used as models, approaches such as TGGs/VTGGs, which directly support *graph patterns* in rules are probably more natural/intuitive for view specification.

4 Formalization of VTGGs as Restricted TGGs

A *VTGG* consists of a triple of metamodels (e.g., Fig. 2) and a set of triple rules (e.g., Fig. 3, Fig. 4, and Fig. 5), which describe the *simultaneous evolution* of

[1] `http://www.eclipse.org/proposals/emf-facet/`

triples of source, correspondence and view models. This induces a *consistency relation* in the following manner: A pair of source and view models are consistent, if and only if a triple consisting of the source and view models connected with a *correspondence model* can be created using rules in the VTGG.

This consistency relation is used to automatically derive *operational* transformations that support incrementally propagating changes to the view back to the source and vice-versa. As VTGGs are designed to optimize the *incremental backward transformation* (view to source), we shall concentrate on this in the following. All other operational transformations can be derived as for standard TGGs and we refer to [8] for arguments concerning formal properties that also apply to VTGGs, which are TGGs with additional restrictions.

After introducing necessary fundamentals on TGGs, we shall first formulate precisely the set of structural restrictions on VTGG rules, and then use these restrictions to argue formal properties of incremental view updates with VTGGs.

4.1 Preliminaries

In the following, we consider VTGGs as TGGs with a set of restrictions on the set of rules. There exists a rich formal foundation for TGGs based on algebraic graph transformation [6] and we refer to, e.g., [8] for further details.

Triple Graph Grammar: A $TGG = (\mathcal{M}, \mathcal{R})$ consists of a triple of metamodels (source, correspondence and target) $\mathcal{M}$, and a set $\mathcal{R}$ of rules. Each rule $r = (L, R) \in \mathcal{R},\ L \subseteq R$ consists of a precondition L and postcondition R, both being triples of source, correspondence and target patterns. As explained in Sect. 2, a rule is applied by mapping the precondition to elements in an input model, i.e., determining a match, which is replaced with the postcondition.

Formal Properties of TGGs: TGG rules describe the simultaneous evolution of triples of models but can be operationalized and used to derive unidirectional forward (source to target) and backward (target to source) transformations. The TGG is also used as a contract that stipulates the expected behaviour of the bidirectional transformation realized with the forward and backward transformations derived from the TGG [8]. In the following we only formulate the laws for the incremental backward transformation as this is what is optimized for VTGGs and must thus be shown to be sound:

Correctness: Correctness demands that the derived backward transformation only creates *consistent* triples, i.e., triples that can be generated by the TGG.

Given a $TGG = (\mathcal{M}, \mathcal{R})$, where $\mathcal{M} = \mathcal{M}_S \leftarrow \mathcal{M}_C \rightarrow \mathcal{M}_T$, let $\mathcal{L}(TGG)$ denote the set of all models that can be derived using rules in $\mathcal{R}$, and $\mathcal{L}(\mathcal{M})$ the set of all model triples that conform to the metamodel triple $\mathcal{M}$.

Let Δ_T denote the set of *all supported changes* $\delta_T = M_T \leftarrow M_T^- \rightarrow M_T^+ \in \Delta_T$, which can be applied to a source model M_T by deleting all elements in $M_T \setminus M_T^-$ and adding all elements in $M_T^+ \setminus M_T^-$.

Let $BT : \mathcal{L}(TGG) \times \Delta_T \rightarrow \mathcal{L}(\mathcal{M})$ denote the *incremental* backward transformation derived from the TGG, a partial function which maps a model triple

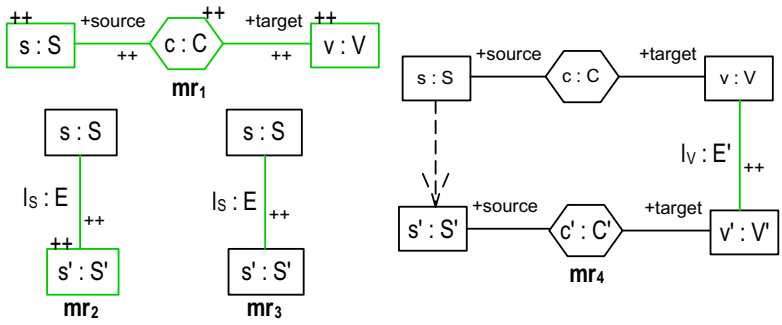


Fig. 7. Meta-rules used to specify the allowed structure of VTGG rules

$M_S \leftarrow M_C \rightarrow M_T \in \mathcal{L}(TGG)$, and a change to the target model $\delta_T = M_T \leftarrow M_T^- \rightarrow M_T^+ \in \Delta_T$, to an updated triple $M'_S \leftarrow M'_C \rightarrow M_T^+ \in \mathcal{L}(\mathcal{M})$.

BT is *correct* iff it produces model triples in $\mathcal{L}(TGG)$, i.e., $range(BT) \subseteq \mathcal{L}(TGG)$.

Completeness: As a backward transformation can be trivially correct by rejecting all input, it is important to demand completeness, i.e., that the derived backward transformation be able to handle *every* input for which there exists a consistent output, i.e, an appropriate triple in $\mathcal{L}(TGG)$.
BT is *complete* iff $\forall\ M_S \leftarrow M_C \rightarrow M_T \in \mathcal{L}(TGG), \forall\ M_T \leftarrow M_T^- \rightarrow M_T^+ \in \Delta_T$, $\exists\ M'_S \leftarrow M'_C \rightarrow M_T^+ \in \mathcal{L}(TGG) \Rightarrow BT(M, \delta_T) \in \mathcal{L}(\mathcal{M})$ is defined.

Efficiency: An *incremental transformation* BT that propagates a source model change by changing the target model incrementally, is efficient if its runtime is independent of the size of the models and depends only on the scope of influence of the change, i.e., all elements that must be re-translated due to the change.

4.2 VTGGs as Restricted TGGs

Given a $TGG = (\mathcal{M} = \mathcal{M}_S \leftarrow \mathcal{M}_C \rightarrow \mathcal{M}_T, \mathcal{R})$, let the target metamodel $\mathcal{M}_T$ be referred to in the following as the *view metamodel*, denoted as $\mathcal{M}_V$. To define the allowed structure of VTGG rules, Fig. 7 depicts four *meta-rules* mr_1 – mr_4. These meta-rules will be used as building blocks to construct the allowed patterns that comprise VTGG rules, substituting the types S, S' with types from the source metamodel, V, V' with types from the view metamodel, and C, C' with types from the correspondence metamodel as required. Meta-rules are depicted using the compact notation (merging L and R), while the actual rules generated by applying meta-rules are built up explicitly by specifying L and R separately.

For a meta-rule mr, let $\mathcal{L}(mr)$ denote the set of triples that can be generated by applying mr on the empty triple (denoted as $\emptyset$) with types from $\mathcal{M}$, i.e.: $\mathcal{L}(mr) = \{M \in \mathcal{L}(\mathcal{M}) \mid \emptyset \overset{mr}{\Rightarrow} M\}$. Given a model M that conforms to a metamodel $\mathcal{M}$, i.e., $M \in \mathcal{L}(\mathcal{M})$, let $M \vdash mr$ mean that M is syntactically defined by mr, i.e., that $M \in \mathcal{L}(mr)$. Furthermore, for meta-rules mr and mr', let

$mr \cdot mr' : \mathcal{L}(mr \cdot mr') = \{M' \in \mathcal{L}(\mathcal{M}) \mid \emptyset \overset{mr}{\Rightarrow} M \overset{mr'}{\Rightarrow} M'\}$ denote their composition. Finally, let mr^* denote an arbitrary composition of mr, i.e., $mr \cdot \ldots \cdot mr$. We now use this notation to define VTGG rules in the following.
$r = (L_C, R_C)$ is a *class rule*[2] iff $L_C = \emptyset, R_C \vdash cr_R$, where:

$$cr_R := mr_1 \cdot mr_2^* \cdot mr_3^* \tag{1}$$

$r = (L_A, R_A)$ is an *association rule* iff $L_A \vdash ar_L, R_A \vdash ar_R$, where:

$$ar_L := cr_R \cdot cr_R, \quad ar_R := ar_L \cdot mr_4 \cdot mr_2^* \cdot mr_3^* \tag{2}$$

The dashed arrow in mr_4 is used to demand that s and s' must be connected by some path in the view model for mr_4 to be applicable.

$r = (L_{IC}, R_{IC})$ is an idle *class rule* iff $L_{IC} \vdash icr_L, R_{IC} \vdash icr_R$, where:

$$icr_L := cr_R, \quad icr_R := icr_L \cdot mr_2^* \cdot mr_3^* \tag{3}$$

$r = (L_{IA}, R_{IA})$ is an *idle association rule* iff $L_{IA} \vdash iar_L, R_{IA} \vdash iar_R$, where:

$$iar_L := ar_R, \quad iar_R := iar_L \cdot mr_2^* \cdot mr_3^* \tag{4}$$

A $VTGG = (\mathcal{M}_S \leftarrow \mathcal{M}_C \rightarrow \mathcal{M}_V, \mathcal{R})$ is a TGG with the following restrictions:

1. Every rule $r \in \mathcal{R}$ is either a class rule, an association rule, or is idle.
2. There is exactly one class rule for every non-abstract view class.
3. There is exactly one association rule for every association in the view metamodel and the context required by the association rule must be guaranteed by the class rules of the source and target classes of the association. Formally, with $r_A = (L_A, R_A)$ an association rule as defined above, and $\uplus$ denoting the disjoint union of graphs:
$$\forall (L_A, R_A) \in \mathcal{R}, \exists (L_C, R_C) \in \mathcal{R}, \exists (L'_C, R'_C) \in \mathcal{R} : L_A \subseteq R_C \uplus R'_C \tag{5}$$
4. The context required by every idle class rule must be guaranteed by the corresponding class rule:
$$\forall (L_{IC}, R_{IC}) \in \mathcal{R}, \exists (L_C, R_C) \in \mathcal{R} : L_{IC} \subseteq R_C \tag{6}$$
5. The context required by every idle association rule must be guaranteed by the corresponding association rule:
$$\forall (L_{IA}, R_{IA}) \in \mathcal{R}, \exists (L_A, R_A) \in \mathcal{R} : L_{IA} \subseteq R_A \tag{7}$$
6. When actually applying association and idle rules in a transformation sequence, only the context guaranteed by (5), (6) and (7) can be used.

Example: After defining the set of VTGG restrictions, we can consider our running example and check if the restrictions hold. The rules depicted in Fig. 3 are indeed class rules as they can be constructed according to (1). For instance, for $r_1 = (L_{r_1}, R_{r_1})$, $L_{r_1} = \emptyset$ and $R_{r_1} \vdash mr_1 \cdot mr_2 \cdot mr_2 \vdash cr_R$. Similarly, the rules depicted in Fig. 4 are association rules as they can be constructed according to (2). For instance, for $r_6 = (L_{r_6}, R_{r_6})$, $L_{r_6} \vdash (mr_1 \cdot mr_2) \cdot mr_1 \vdash cr_R \cdot cr_R \vdash ar_L$ and $R_{r_6} \vdash ar_L \cdot mr_4 \cdot mr_3 \vdash ar_R$. Furthermore, $L_{r_6} \subseteq R_{r_3} \uplus R_{r_2}$ as required

[2] Note that $L_C \subseteq R_C$ must hold as a class rule is a rule.

by (5). Finally, the rules in Fig. 5 are idle as they can be constructed according to (3) and (4). For instance, for $r_9 = (L_{r_9}, R_{r_9})$, $L_{r_9} \vdash cr_R \vdash icr_L$ and $R_{r_9} \vdash icr_R \cdot mr_2 \vdash icr_R$. Furthermore, $L_{r_9} \subseteq R_{r_1}$ as required by (6).

VTGGs Are Correct: To show correctness for VTGGs, we have to prove that incremental view updates (view to source) via BT_{VTGG}, denoting the optimized incremental backward transformation for VTGGs, are consistent with the VTGG. For a given model triple $M = M_S \leftarrow M_C \rightarrow M_V \in \mathcal{L}(VTGG)$ and a change to the view $\delta_V \in \Delta_V$, we do this constructively by stating how the following cases (corresponding to the four types of supported changes) must be handled:

(1) Object Creation: As there must exist exactly one class rule r_V for every class V in the view metamodel, creating an object o_V of type V in the view corresponds to applying r_V to create o_V and the corresponding source structure as defined in the rule. As the given model triple M is in $\mathcal{L}(VTGG)$, there exists a sequence of VTGG rule applications $r_1, r_2, \ldots, r_k$ that can be used to create M. As the class rule r_V, according to (1), is of the form $(\emptyset, R_V)$ it is independent of all other rules and can be applied to extend the sequence to $M' = r_1, r_2, \ldots, r_k, r_V$. The resulting model triple M' is, therefore, in $\mathcal{L}(VTGG)$ and is consistent.

(2) Link Creation: As there must exist exactly one association rule r_A for every association A in the view metamodel, creating a link l_A of type A in the view model between two objects o_V and $o_{V'}$ of type V and V', respectively, corresponds to applying the association rule r_A. As o_V and $o_{V'}$ already exist in the view, they can only have been created by applying the corresponding class rules r_V and $r_{V'}$, respectively. As the VTGG restriction (2) guarantees that the required context (which *must* be used!) for applying r_A is implied by the applications of r_V and $r_{V'}$, the sequence of rule applications $M = r_1, \ldots, r_V, \ldots, r_{V'}, \ldots, r_k$ is extended to $M' = r_1, \ldots, r_V, \ldots, r_{V'}, \ldots, r_k, r_A$, implying correctness. Note that r_A is only allowed to depend on r_V and $r_{V'}$.

(3) Link Deletion: To correctly propagate deletion of a link l_A in the view model, the *scope of influence* of the change must be determined. In the case of link deletion, only idle association rule applications r_{IA} can depend on source elements created by the corresponding association rule r_A according to (4). As no other rule applications can depend on idle association rule applications, the latter can be safely reverted by deleting all created elements in the source. After reverting all dependent idle association rule applications, the link deletion can be propagated by reverting the application of the association rule r_A, deleting the link in the view and the corresponding source structure. The remaining sequence of rule applications is valid (a possible sequence of VTGG rule applications).

(4) Object Deletion: The scope of influence of deleting an object in the view comprises all association rule applications that created incident links to the deleted object and all idle class rule applications that require the deleted object as context. To delete an object in the view, therefore, all incident links must be deleted and appropriately propagated to the source model according to *Link*

Deletion, then all dependent idle class rule applications are reverted, before finally reverting the class rule application used to create the object to be deleted. As all dependent rule applications are thus reverted, the remaining sequence of rule applications is valid and propagation of object deletion is correct.

VTGGs Are Complete: The four cases distinguished above also define Δ_T, the set of supported changes to the view. As the arguments for correctness describe constructively how the allowed changes are to be propagated for an arbitrary model triple, it follows that *every* allowed change can be propagated successfully in this manner. BT_{VTGG} is, therefore, complete.

VTGGs Are Efficient: BT_{VTGG} is efficient, as the VTGG restrictions guarantee a local scope of influence (defined above for correctness of BT_{VTGG}). Propagating changes, therefore, depends only on elements in the scope of influence of the change, which is, by construction, independent of the size of the models.

5 Runtime Measurements

Our runtime results, depicted in Fig. 8, were obtained by measuring the time required to create and add a UIPart to randomly generated view models of increasing size (1000 – 30,000 elements). The VTGG rules for our running example were used to generate a TGG batch transformation (blue curve), a TGG-based synchronizer (red curve), and a VTGG-based implementation, all with the same model transformation tool eMoflon (`www.emoflon.org`). The exact same update was executed 11 times (the median is shown in the plot) for each model size on a PC with an Intel(R) Core(TM)2 Duo 2.53GHz CPU, and 8GB RAM, running Windows 7 (64 Bit), Oracle JDK 1.7.007 and Eclipse 4.3.1. Although this is only one rather simple example with only a handful of rules, our results nonetheless

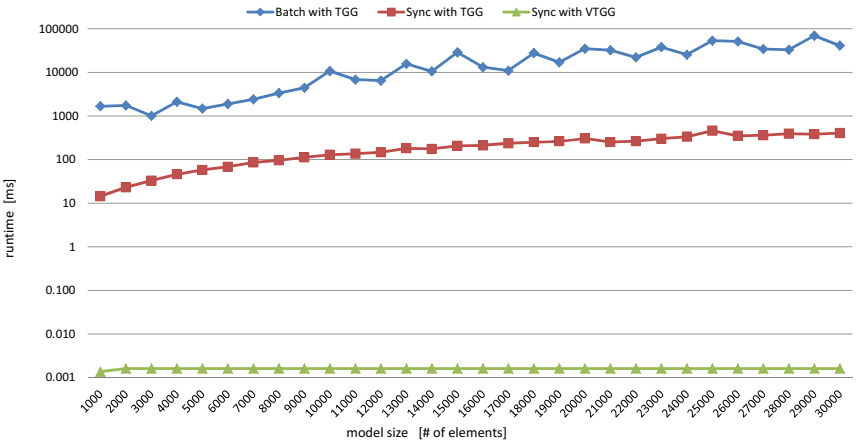


Fig. 8. Runtime measurements for running example

indicate that VTGGs have the potential of being magnitudes (μs as compared to ms for 30,000 elements) faster than the current standard TGG synchronization implemented according to [8]. Even more importantly, VTGGs are truly efficient in the sense of [8], meaning that the time required for synchronization is completely independent of the actual models size.

6 Conclusion and Future Work

In this paper we have formalized a view specification framework based on VTGGs. Motivated with a simplified but real-world application, we explained the necessary restrictions that enable an efficient implementation. Implementation details of our current VTGG implementation concerning, e.g., auxiliary data required to track rule dependencies and revert rule applications correctly and efficiently, detecting changes to the view via an appropriate notification framework, and the operationalization of attribute conditions in VTGG rules were out-of-scope for this paper and are currently being investigated and evaluated in detail. Possible extensions for VTGGs include: (i) reducing restrictions on class and association rules as a trade-off between expressiveness and formal guarantees, (ii) integrating ideas from [7] to support non-materialized views in cases where this is required, and (iii) extending our concept of views to other aspects such as methods.

References

1. Anjorin, A., Varró, G., Schürr, A.: Complex Attribute Manipulation in TGGs with Constraint-Based Programming Techniques. In: BX 2012. ECEASST, vol. 49, pp. 1–15. EASST (2012)
2. Bohannon, A., Foster, J., Pierce, B., Pilkiewicz, A., Schmitt, A.: Boomerang: Resourceful Lenses for String Data. ACM SIGPLAN Notices 43(1), 407–419 (2008)
3. Bohannon, A., Pierce, B.C., Vaughan, J.A.: Relational Lenses: A Language for Updatable Views. In: PODS 2006, pp. 338–347. ACM (2006)
4. Diskin, Z., Xiong, Y., Czarnecki, K.: From State- to Delta-Based Bidirectional Model Transformations: The Asymmetric Case. JOT 10, 1–25 (2011)
5. Ehrig, H., Ehrig, K., Ermel, C., Prange, U.: Consistent Integration of Models based on Views of Meta Models. Formal Aspects of Computing 22(3-4), 327–344 (2010)
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation, 1st edn. Springer (2006)
7. Jakob, J., Königs, A., Schürr, A.: Non-Materialized Model View Specification with Triple Graph Grammars. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 321–335. Springer, Heidelberg (2006)
8. Lauder, M., Anjorin, A., Varró, G., Schürr, A.: Efficient Model Synchronization with Precedence Triple Graph Grammars. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 401–415. Springer, Heidelberg (2012)
9. Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A.: Xlinkit: A Consistency Checking and Smart Link Generation Service. ACM Transactions on Internet Technology 2(2), 151–185 (2002)

10. Nuseibeh, B., Kramer, J., Finkelstein, A.: ViewPoints: Meaningful Relationships are Difficult!. In: Clarke, L.A., Dillon, L., Tichy, F.W. (eds.) ICSE 2003, pp. 676–683. IEEE (2003)
11. Ranger, U., Gruber, K., Holze, M.: Defining Abstract Graph Views as Module Interfaces. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 120–135. Springer, Heidelberg (2008)
12. Yokoyama, T., Axelsen, H., Glück, R.: Principles of a Reversible Programming Language. In: Ramírez, A., Bilardi, G., Gschwind, M. (eds.) CF 2008, pp. 43–54. ACM (2008)