# Model View Management with Triple Graph Transformation Systems

Esther Guerra[1] and Juan de Lara[2]

[1] Computer Science Department,
Universidad Carlos III de Madrid (Spain)
eguerra@inf.uc3m.es
[2] Polytechnic School,
Universidad Autónoma de Madrid (Spain)
jdelara@uam.es

**Abstract.** In this paper, we present our approach for model view management in the context of Multi-View Visual Languages (MVVLs). These are made of a number of diagram types (or viewpoints) that can be used for the specification of the different aspects of a system. Therefore, the user can build different *system views* conform to the viewpoints, which are merged in a repository in order to perform consistency checking. In addition, the user can define *derived views* by means of *graph query patterns* in order to extract information from a base model (a system view or the repository). We have provided automatic mechanisms to keep synchronized the base model and the derived view when the former changes. Predefined queries by the MVVL designer result in so-called *audience-oriented views*. Finally, *semantic views* are used for analysing the system by its translation into a semantic domain.
Our approach is based on meta-modelling to describe the syntax of the MVVL and each viewpoint, and on triple graph transformation systems to synchronize and maintain correspondences between the system views and the repository, as well as between the derived, audience-oriented and semantic views and the base models. We illustrate these concepts by means of an example in the domain of security for web systems.

## 1 Introduction

Visual Languages (VLs) are extensively used in many engineering activities for the specification and design of systems. As these become more complex, there is a need to split specifications into smaller, more comprehensible parts that use the most appropriate notation. Thus, there are VLs (such as UML) made of a family of diagram types, which can be used for the description of the different aspects of a system. We call such VLs Multi-View Visual Languages (MVVLs) [5]. In these languages, the user has a need of building models (using the different diagram types) and check their consistency; of querying models to obtain partial models; and of transforming models into other formalisms. All these artefacts can be considered different kinds of views of the system. This necessity has been

recently recognised by the OMG by defining a standard language for expressing Queries, Views and Transformations (QVT, see [11]).

Our approach for model view management is based on meta-modelling and graph transformation. With meta-modelling, we define the syntax of the complete MVVL. Each diagram type (or viewpoint) has its own meta-model too, which is a part of the complete MVVL meta-model. At the model level, the user builds different *system views* conform to a viewpoint meta-model. System views are merged together in a unique model called *repository*. Triple Graph Transformation Systems (TGTSs) automatically derived from the meta-models perform the merging, allow incremental updates and relate the system views and the repository. They also provide syntactic consistency and change propagation from one view to the others (i.e. they are bidirectional). In addition, it is possible to generate TGTSs modelling different behaviours for view management (e.g. cascading deletion vs. conservative deletion of elements).

We also present *graph query patterns* as a declarative visual query language to obtain *derived views* (in the sense of QVT [11]) from a base model. Starting from the patterns, a TGTS is automatically generated to build the derived view and maintain it consistent with respect to changes in the base model (i.e. derived views are incremental). If the query is predefined by the MVVL designer and later used by a specific kind of user, we call it *audience-oriented view*.

Finally, the system views (or the repository) can be translated into another formalism for dynamic semantics checking, analysis and simulation. We call the target model *semantic view*. The MVVL designer defines the translation by means of a TGTS that establishes correspondences between the elements in the source model and its semantic view. Thus, the results of the analysis can be back annotated and shown in the base model, likely more intuitive for the user.

The main contribution of this paper is the use of a uniform specification of the different kinds of views by means of meta-modelling and TGTSs. Moreover, we propose graph query patterns to specify derived views, together with mechanisms to automatically obtain TGTSs that build the view. In [5] we presented the first steps towards the definition of MVVLs, where we only considered system views. We have extended previous work, as we now consider other kinds of views, and improve system views by allowing configurable behavioural patterns. This work is founded in an extension of the classical notion of Triple Graph Grammars (TGGs) by Schürr [13]. In its original sense, a TGG is a grammar that generates a language of triple graphs, from which triple rules implementing forward or backward translations are derived. In our case, we generate the TGTSs that implement translations and propagation of updates from meta-models or queries. In addition, our TGTSs are formally defined in the double pushout approach (DPO), and extend triple graphs with inheritance and more flexible morphisms in the correspondence graph (see [6] for details).

The paper is organized as follows. Section 2 presents an overview of our formalization of TGTSs. Section 3 shows our approach for defining the syntax of MVVLs and handling system views. In subsection 3.1, we describe several ways of configuring the behaviour of a modelling environment to manage the system

views. Section 4 describes graph query patterns and how TGTSs to build the derived view are obtained from them. Section 5 shows how to define a semantic view. Section 6 compares with related research. Finally, section 7 ends with the conclusions and further research. In all sections, we illustrate the concepts with an example of MVVL in the domain of security for web systems.

## 2 Triple Graph Transformation Systems

TGTSs model transformations of triple graphs, which are made of three separate graphs: source, target and correspondence. As originally defined [13], nodes in the correspondence graph had morphisms to nodes in the source and target graphs. We have extended the notion of triple graph by allowing attributes on nodes and edges. Moreover, the relation between the source and target graphs is more flexible, as we allow morphisms from nodes in the correspondence graph to nodes and edges in the other two graphs, as well as being undefined. Finally, we also provide triple graphs with a typing by a triple type graph (similar to a triple meta-model) which may contain inheritance relations between nodes or edges. We follow the DPO approach [3] for the formalization of triple graph rules (see [6] for details). Here, for space limitations, we only present a brief summary.

Our triple graphs are based on the notion of $E - graph$ [3], which extends regular graphs with node and edge attribution. Attribute values are indeed data nodes, while attributes are edges connecting graph nodes and edges with data nodes. We define a $TriE - graph$ as three E-graphs (source, correspondence and target) and two correspondence functions $c_1$ and $c_2$. The correspondence functions are defined from the nodes in the correspondence graph to nodes and edges in the other two graphs. In addition, the functions can be undefined. This is modelled with a special element in the codomain (named "·"). Visually, this is denoted as a correspondence graph node from which some of the correspondence functions are missing (see for example rules in Fig. 7). TriE-graph objects and morphisms form category **TriEGraph**. In order to structure the data values in sorts and make operations available, TriE-graphs are provided with an algebra over a suitable signature, resulting in category **TriAGraph**. Finally, we provide TriAGraphs with a typing by defining a triple type graph (similar to a meta-model triple). This is an attributed triple graph where the algebra is final. The typing is a TriAGraph-morphism from the graph to the type graph. Indeed, attributed typed triple graphs (short ATT-graphs) can be modelled as objects in the slice category **TriAGraph/TriATG**, which we write as $\mathbf{TriAGraph_{TriATG}}$. In [6] we extend type graphs and triple graph rules with inheritance for nodes and edges.

Fig. 1 shows an ATT-graph that relates a source Role Based Access Control model [4] (up) and a target Coloured Petri Net [9] (down). Its meta-model triple is not shown in the paper for space constraints, although the part that corresponds to the source type graph is shown on the upper part of Fig. 4.

We manipulate ATT-graphs by means of DPO triple rules. In the DPO approach, rules are modelled using three components: $L$, $K$ and $R$. The $L$ compo-

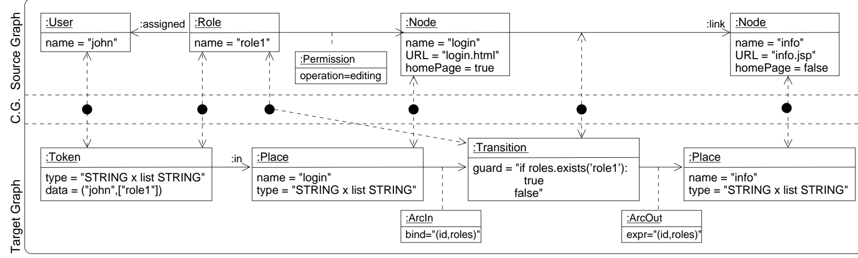



**Fig. 1.** An Attributed Typed Triple Graph.

nent (or left hand side, LHS) contains the elements to be found in the structure (a graph, a Petri net, etc.) where the rule is applied. $K$ (the kernel) contains the elements preserved by the rule application. Finally, $R$ (the right hand side, RHS) contains the elements that should replace the part identified by $L$ in the structure. Therefore, $L - K$ are the elements that should be deleted by the rule application, while $R - K$ are the elements that should be added. Note that DPO transformation has been lifted to work not only in the **Graph** category, but also with any (weak) adhesive HLR category [3]. In [6] we show that category $\mathbf{TriAGraph_{TriATG}}$ is an adhesive HLR category. Therefore, in our case, $L$, $K$ and $R$ are ATT-graphs.

In addition, we provide triple rules with a set of application conditions that restrict their applicability. An application condition $c = (X, X \xrightarrow{y_i} Y_i)$ has a premise ATT-graph $X$, a set of consequent ATT-graphs $Y_i$, and morphisms $y_i$ from $X$ to each $Y_i$. In order to apply a rule, if a match of $X$ is found in the host ATT-graph, then a match of some $Y_i$ has also to be found. If the application condition does not have any consequent ATT-graph, finding a match of the premise forbids the rule application. This is a special case of condition called *negative application condition* (NAC). On the contrary, if the premise is isomorphic to the LHS, then this is a *positive application condition* (PAC). We use a shortcut notation for application conditions: the subgraph of $L$ (resp. $X$) that does not have an image in $X$ (resp. $Y_i$) is isomorphically copied into $X$ (resp. $Y_i$) and appropriately linked with their elements.

Fig. 2 shows a triple rule. It creates a place in the target graph (lower part) and relates it with an existing node in the source graph (upper part). The $NAC$ forbids the rule application if the node is related to a place. The $K$ component is omitted for clarity. It contains the common elements of $LHS$ and $RHS$ (i.e. node labelled "1"). We will use such notation throughout the paper.

## 3 Multi-View Visual Languages: System Views

MVVLs are made of a set of diagram types, each one of them defined by its own meta-model and dealing with a different viewpoint of the system [5]. However, all these separate definitions are based on a unique meta-model that relates

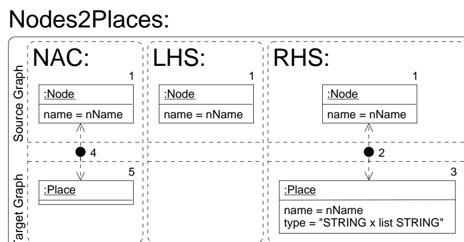



**Fig. 2.** An Attributed Typed Triple Graph Rule.

their abstract syntax concepts. This is for example the approach of UML2.0. The different viewpoints may overlap in this unique meta-model. It is key to identify the overlapping parts for each pair of viewpoints in order to be able to maintain them coherent through their common elements. Fig. 3 expresses this situation in categorical terms. Thus, a MVVL is defined by means of an attributed type graph $TG^{MVVL}$ (i.e. its meta-model). The different viewpoints $TG^{VP}$ are inclusions of it, although in a more general approach they can be any function. For each two viewpoints $TG^{VP_i}$ and $TG^{VP_j}$, the overlapping part $I^{i,j}$ is calculated as the pullback of its respective type graphs and $TG^{MVVL}$. Thus, $id_i \circ o_{i,j} = id_j \circ o_{j,i}$. At the model level, the user builds *system views* conform to some viewpoint (i.e. a typing morphism exists from each system view to a viewpoint). Note how there might be more than one system view for the same viewpoint. In order to guarantee syntactic consistency, a *repository model* is built by merging all the system views. The repository is the colimit of the views, and there is a typing morphism from it to the MVVL type graph.

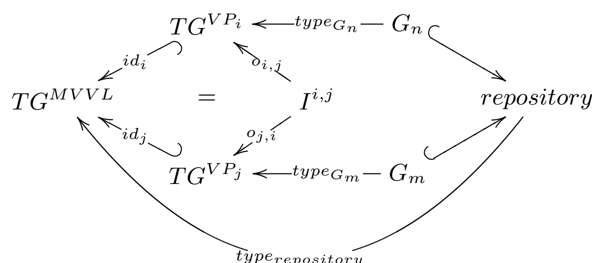



**Fig. 3.** Multi-View Visual Language Definition in Categorical Terms.

In our approach, the merging operation is performed by TGTSs automatically derived from the meta-model information. Note how updating the repository (because of a system view modification) may leave other views in an inconsistent state. At this point, other automatically generated TGTSs update the rest of the views. The identification of the overlapping of each two viewpoints helps to minimize the rules that must be tried in this change propagation. In this way,

we have TGTSs that propagate changes in the two directions in consecutive steps: first from the views to the repository, and then the other way round (if necessary). This is similar to the Model-View-Controller pattern. For static semantics consistency the MVVL designer may provide additional triple rules. In this way, both syntax and static semantics can be checked in a uniform way.

In order to illustrate these concepts, as well as the ones presented in the following sections, we introduce a case study for modelling a Role Based Access Control (RBAC) [4] for web systems[3]. Its meta-model is shown in the upper part of Fig. 4. Briefly, a web application is made of nodes with a name and a URL, where one of the nodes is the home page. Navigation between nodes is modelled by means of relation "link". In addition, roles can be defined with a set of permissions for accessing nodes. Roles are nested in hierarchies through relation "contains", and inherit all permissions of reachable roles through such relation. Finally, users can be assigned a set of roles, from which they obtain the permissions for interacting with the system.

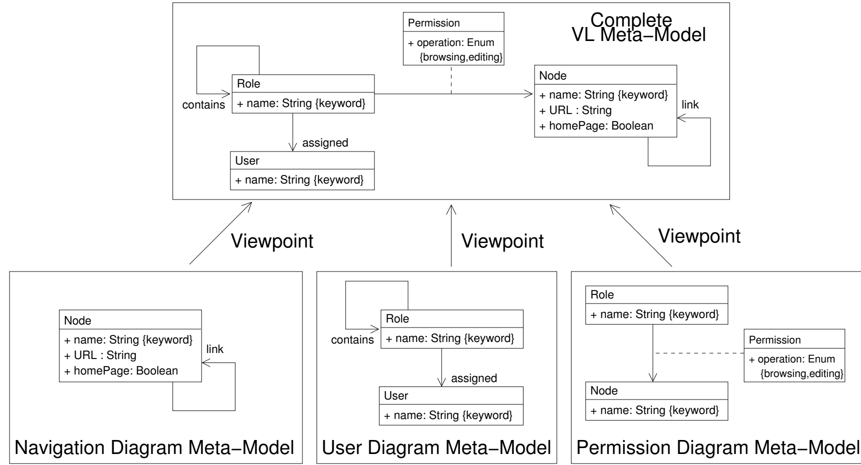



**Fig. 4.** Definition of a Role Based Access Control for Web-Based Systems.

This meta-model comprises the structure of the web system as well as its security policy. However, it is more suitable to handle each aspect in separate diagrams. Thus, we specify three viewpoints on the MVVL meta-model. The first one (*Navigation Diagram*) is used for specifying the web structure, and it only contains nodes and links. The second one (*User Diagram*) is used for specifying role hierarchies and their assignment to users. It contains roles, users, and relations "contains" and "assigned". The third viewpoint (*Permission Diagram*) allows assigning permissions to roles. It contains roles, nodes and permissions.

[3] We do not consider the internal structure of web pages for simplicity.

Note how in this view we are only interested in identifying the node, therefore only its attribute "name" is relevant.

From this definition, a TGTS is automatically generated between each viewpoint and the repository. Their purpose is building the repository model from the system views. The transformation systems contain the rules shown in Fig. 5 for each concrete class and association[4] in the viewpoint. We only show the rules for class "Role". The first creation rule adds a role to the repository if, given a new role in a view, it does not exist in the repository yet (NAC1). NAC2 is needed to prevent creating a new role in the repository when changing the name of an existing role in the view (i.e. with an associated role in the repository). The second creation rule relates a new role in a view with the same role in the repository. This means that the role was previously created and added to the repository from another view. Attribute "refcount" counts how many times a role appears in different views. When a role is added to the repository, the counter is set to 1; each time the same role is added to any view, the counter is incremented. The first deletion rule detects when a role is removed from a view (i.e. the correspondence function to the view is undefined for a role in the repository), and decrements the "refcount" attribute. When the counter reaches zero, this means that the role does not appear in any view, so it is removed from the repository by the second deletion rule. Finally, the editing rule propagates changes in the attributes from the roles in the views to the corresponding roles in the repository.

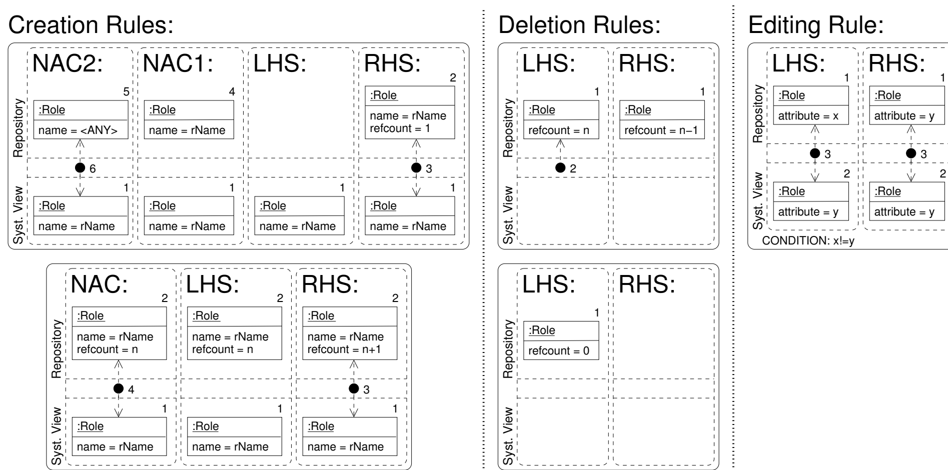



**Fig. 5.** Triple Graph Rules for Building the Repository.

[4] An association is not unambiguously identified by its source and target. Thus, we relate associations in views with the corresponding ones in the repository for change propagation. This is possible as we allow correspondence functions to edges [6].

Other TGTSs (one for each viewpoint) are automatically generated which propagate changes in attribute values from the repository to the other system views. These are made of just one kind of rule, like the one shown in Fig. 6. These transformation systems are applied only when the repository has changed due to the execution of one of the former TGTSs.

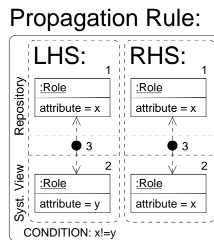



**Fig. 6.** Triple Graph Rules for Change Propagation.

### 3.1 Configurable Behavioural Patterns

The presented TGTSs determine the behaviour of the MVVL modelling tool. However, a different behaviour can be more appropriate for a given MVVL. For this reason, we provide a catalogue of different behavioural patterns to configure the behaviour of the modelling tool. Different sets of automatically generated rules are added to the consistency TGTSs depending on the desired behaviour.

For example, the deletion rules in Fig. 5 perform a conservative deletion (i.e. a role is deleted from the repository only when the user has deleted all its instances from the views). On the contrary, cascading deletion implies that when an element is removed from a view, it is also removed from any other view and the repository. Such behaviour can be provided by replacing the previous deletion rules by the ones shown in Fig. 7. The first two rules belong to the TGTS for building the repository. The first one detects when a role has been removed from a view (i.e. the correspondence function to the view for the role in the repository is undefined), and then removes it from the repository. The dangling condition forbids the rule application when the role has incoming or outgoing relations. The second rule handles the deletion of one of such relations. Similar rules are generated for each possible incoming and outgoing relation to a role. Note how the rule is applied to the triple graph relating a particular view and the repository; therefore, correspondence relations from the same node to elements in other views are not part of the graph.

Other rules propagate the deletion of elements from the repository to the other views. Some of them are shown to the right of Fig. 7. The first rule deletes a role from a view if it is not in the repository. The dangling condition forbids the rule application if the role has some incoming or outgoing relation. The following

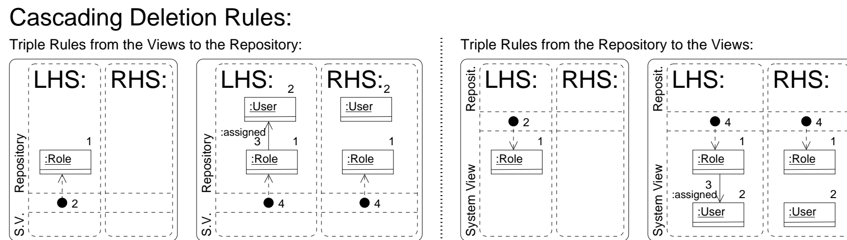



**Fig. 7.** Cascading Deletion by means of Triple Graph Rules.

rule (and other similar ones for each possible type of relation) handles this. It can be noted how this behavioural pattern does not need attribute “refcount”.

The presented TGTSs are asynchronous. They are executed when the user finishes editing a system view and validates the changes. In addition, there are also synchronous behavioural patterns that execute a TGTS in response to a user action (e.g. a creation). One example is the behavioural pattern for intelligent creation. This creates one rule for each type in the complete MVVL meta-model that is immediately executed after creating a new element of such type in a view and specifying its keyword. The rule copies the value of the attributes from the same element in the repository (if exists) to the element in the view.

## 4 Derived and Audience-Oriented Views

In addition to adding information to the system, there is a necessity of extracting information from it. In this way, a *derived view* is defined as a sub-model that contains part of another model, called *base model*. The users of a modelling tool can define derived views. However, there is also the possibility for the MVVL designer to predefine derived views oriented to certain type of final user. We call them *audience-oriented views*. For both kinds of views, we propose the use of a kind of declarative, visual queries called *graph query patterns*. These are evaluated on a base model $G$ (a system view or the repository) to obtain the derived view $V^Q$. A query pattern $Q = (TG^Q, \{(P_i^Q, P_i)\}_{i \in I}, \{(N_j^Q, N_j)\}_{j \in J})$ is made of:

- a meta-model $TG^Q$. It is a restriction of the base model's meta-model $TG$.
- a set of positive restrictions $P_i$. They are patterns that have to be present in the base model for an element to be included in the derived view. $P_i^Q$ contains the element of $P_i$ to which the restriction is applied. Whereas $P_i$ is typed over $TG$, $P_i^Q$ is typed over $TG^Q$. Several positive restrictions applied on the same type have a meaning of “or”; if the restrictions apply on different types, they have a meaning of “and”.
- a set of negative restrictions $N_j$. They are patterns that must not be fulfilled by the elements included in the derived view. As before, a subgraph $N_j^Q$ contains the element where the restriction is applied. Several negative

restrictions applied on the same or different types have a meaning of "and" (all have to be fulfilled).

Fig. 8 shows a diagram with a query pattern evaluated on a base model $G$ (the typing of the restriction graphs has been omitted for clarity). In a first step, the pullback object $V^G$ of $id_{TG^Q}: TG^Q \rightarrow TG$ and $type_G: G \rightarrow TG$ is calculated (square (1) in the figure). $V^G$ is the restriction of $G$ by meta-model $TG^Q$. In a second step, graph $V^G$ is further restricted to take into account the restrictions of the query pattern. Thus, for each match $p_{il}$ of a positive restriction $P_i$, the element identified by $P_i^Q$ has to be included in $V^Q$, such that square (2) commutes[5]. Moreover, if a matching $n_{jk}^Q$ from a negative restriction $N_j^Q$ is found on $V^Q$, then no matching $n_{jk}$ must be found from $N_j$ to $G$ such that square $id_{V^G} \circ id_{V^Q} \circ n_{jk}^Q = n_{jk} \circ id_{N_j^Q}$ commutes. Thus, for an element $x$ with type $type_{V^Q}(x)$ to be included in $V^Q$, it has to fulfill some of the positive restrictions and all the negative ones defined in the query pattern for such type[6].

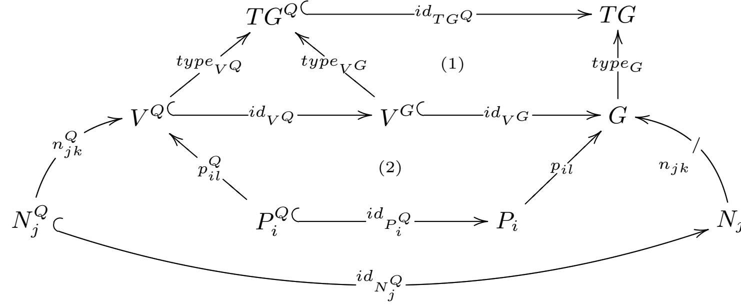
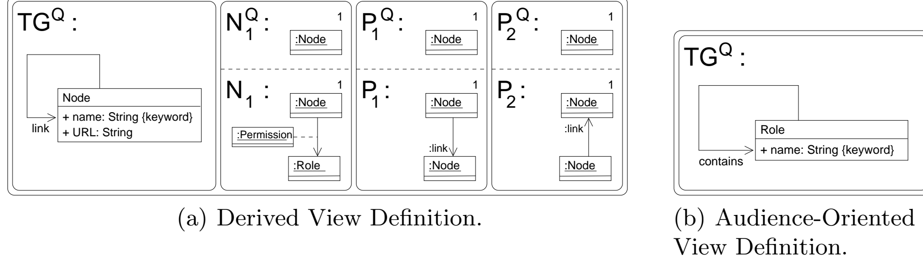



**Fig. 8.** Query Pattern Evaluated on graph $G$ to obtain the Derived View $V^Q$.

As an example, Fig. 9(a) shows a graph query pattern to be executed on the repository model. It defines a derived view, which should contain nodes and links (modelled by graph $TG^Q$ in the query pattern). The derived view should include nodes for which no role is allowed to access (negative restriction $N_1$), and which are source or target of a link (positive restrictions $P_1$ and $P_2$). That is, the derived view contains those nodes that are not isolated in the navigation design but for which nobody has been granted access permission. On the other hand, Fig. 9(b) contains a graph query pattern defining an audience-oriented view with the role hierarchy extracted from the repository. In this case, it is enough to express the $TG^Q$ component, since no additional restriction is imposed.

[5] In general, $P_i^Q$ is not the pullback object of $TG^Q$, $P_i$ and $TG$

[6] By now, we only allow subgraphs $P_i^Q$ and $N_j^Q$ to contain either a node for specifying restrictions on the node, or an arrow between two nodes for specifying restrictions on the arrow. Restrictions that apply to more complex graphs are up to future work.




(a) Derived View Definition.

(b) Audience-Oriented View Definition.

**Fig. 9.** Graph Query Patterns for the Example.

In order to evaluate a graph query pattern on a base model, a TGTS is automatically generated from the pattern. This TGTS creates the derived view, as well as correspondences between its elements and the base model elements. Afterwards, if the base model changes, the TGTS also propagates the changes to the derived view, taking into account the negative and positive restrictions.

Formally, given a query pattern $Q$ to be applied on a base model typed over $TG$, the triples rules generated for it can be expressed as $TGTS(Q, TG) = \{C^N, C^E, E^N, E^E, D^N, D^E\}$. Set $C^N$ (resp. $C^E$) contains the rules that copy the nodes (resp. edges) of the types in $TG^Q$ from the base model to the derived view. Sets $E^N$ and $E^E$ contain the rules that copy the attributes from nodes and edges in the base model to the derived view. Finally, $D^N$ and $D^E$ contain rules that delete nodes and edges from the derived view when they are removed from the base model, or when they (or their context) change and are not consistent with the query pattern restrictions. Positive restrictions in the query pattern are transformed into PACs of the rules in $C^i$, and into NACs of additional deletion rules in $D^i$. More precisely, an extra deletion rule is added to $D^i$ for each set of positive restrictions applied on the same type. On the contrary, negative restrictions are transformed into NACs of the rules in $C^i$, and into PACs of additional deletion rules in $D^i$. This time one deletion rule is created for each negative restriction, independently of the type where the restriction is applied. Due to the simplicity of our rules, we can easily translate the graph constraints into application conditions [7].

Fig. 10 shows the derived rules for class “Node” from the query pattern in Fig. 9(a). The lower part of the rules corresponds to the base model, and the upper part to the derived view to be created. Creation rules, as the one in the figure, always contain the $LHS$, $RHS$ and $NAC1$ by default. In addition, for this example, the positive restrictions $P_1$ and $P_2$ are transformed into the application condition $(X3, X3 \rightarrow Y3_1, X3 \rightarrow Y3_2)$. Thus, a node is added to the derived view only if it is source $(Y3_1)$ or target $(Y3_2)$ of a link. Besides, the negative restriction $N_1$ is transformed into the application condition $NAC2$. Thus, a node is not added to the view if some role can access it. The editing rule simply copies the value of the attributes from the nodes in the repository to the nodes in the derived view.

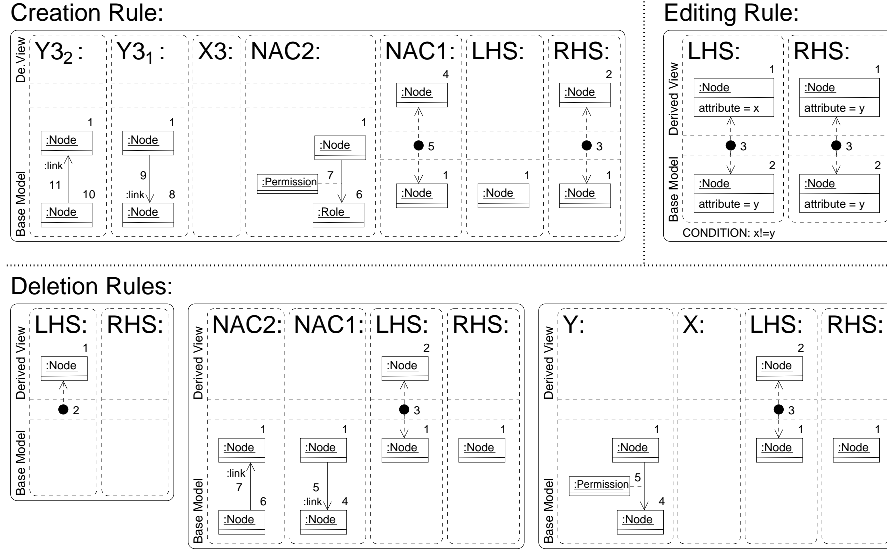



**Fig. 10.** Triple Graph Transformation System Derived from a Graph Query Pattern.

Creation and editing rules are enough for building the derived view. However, a subsequent change in the base model may produce the addition of an element in the view if it fulfills the query pattern; or its deletion if it does not fulfill the query pattern anymore. The former is taken into account by the creation and editing rules. The latter is provided by the deletion rules. The first deletion rule in Fig. 10 removes a node from the view if it does not appear in the base model (i.e. the correspondence function to the base model is undefined). The second rule is derived from the positive restrictions $P_1$ and $P_2$. In this way, if a node is neither source nor target of a link, but it appears in the view, it has to be removed from it. The third rule is derived from the negative restriction $N_1$. Note how a change in any system view is propagated by the consistency TGTSs (Fig. 5) to the repository, and from there, to the derived view by this TGTS.

Fig. 11 shows the derived view that results from the application of the query pattern in Figure 9(a) to a repository model. It contains all the repository nodes except “login” (since it does not satisfy the negative restriction, that is, a role has an editing permission on it) and “admin” (since it does not satisfy the positive restrictions, that is, it is neither source nor target of a link). The links between the nodes are also copied to the view, since they were specified in the $TG^Q$ component of the query pattern. No other types are copied.

## 5 Semantic Views

*Semantic views* are parts of the system expressed in other formalism for dynamic semantics checking, analysis and simulation. With this purpose, the MVVL de-

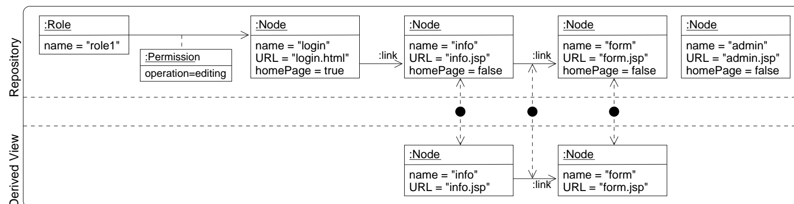



**Fig. 11.** Derived View from a Repository Model.

signer can define a TGTS to generate the target model (or semantic view) from a source model (usually the repository, but also audience-oriented and system views). This allows keeping correspondences between the elements of both models, in such a way that the results of analysing the semantic model could be back annotated to the source model.

For example, rules in Figs. 2 and 12 form a TGTS that generates a semantic view of the repository by translating it into a Coloured Petri Net (CPN) [9]. Thus, source graphs in the rules (up) have to conform to the complete MVVL meta-model shown in Fig. 4. Target graphs (down) have to conform to a CPN meta-model, not shown in the paper for space constraints. This contains places with a type, transitions with a guard, tokens with a type and data value, and arcs from places to transitions and the other way round (the former with a binding of the token data values to variables, and the latter with an expression that is evaluated on such data values to change its value). There is also a meta-model for the correspondence graph, which is not shown.

Rule *Users2Tokens* creates a token for each user. The token has type $STRING \times list\ STRING$ and stores the user name and a list with its assigned roles (initially empty). The NAC forbids multiple applications of the rule for the same user. Next, rule *Assignments2DataValues* builds such list of assigned roles. Thus, each role assigned to a user is added to the second component of the data value of the corresponding token. Rule *Nodes2Places* in Fig. 2 creates a place for each node in the source model. Its type is $STRING \times list\ STRING$, since it will contain tokens of such type. Rule *UsersAtHome* puts tokens into the place that corresponds to the homepage. Rule *Links2Transitions* translates each link between two nodes into a transition between the places that correspond to the nodes. The incoming arc to the transition binds the type of tokens to the variables *id* and *roles*. Their value does not change when the transition is fired. However, the transition has a guard that forbids tokens without certain permissions to pass through. Note how, in this rule we associate nodes of type *Transition* with edges of type *link*. Rule *Permissions2Guards* builds the guard expression. Thus, each permission given to a role to access a node is transformed into a condition that is evaluated to true in the guard. If a token has one of the roles allowed by the transition, it can pass through. Finally, rule *InheritedPermissions2Guards* allows inheritance of permissions. Thus, if a role contains another role that can access certain node, then the former can access it as well. The rule modifies the

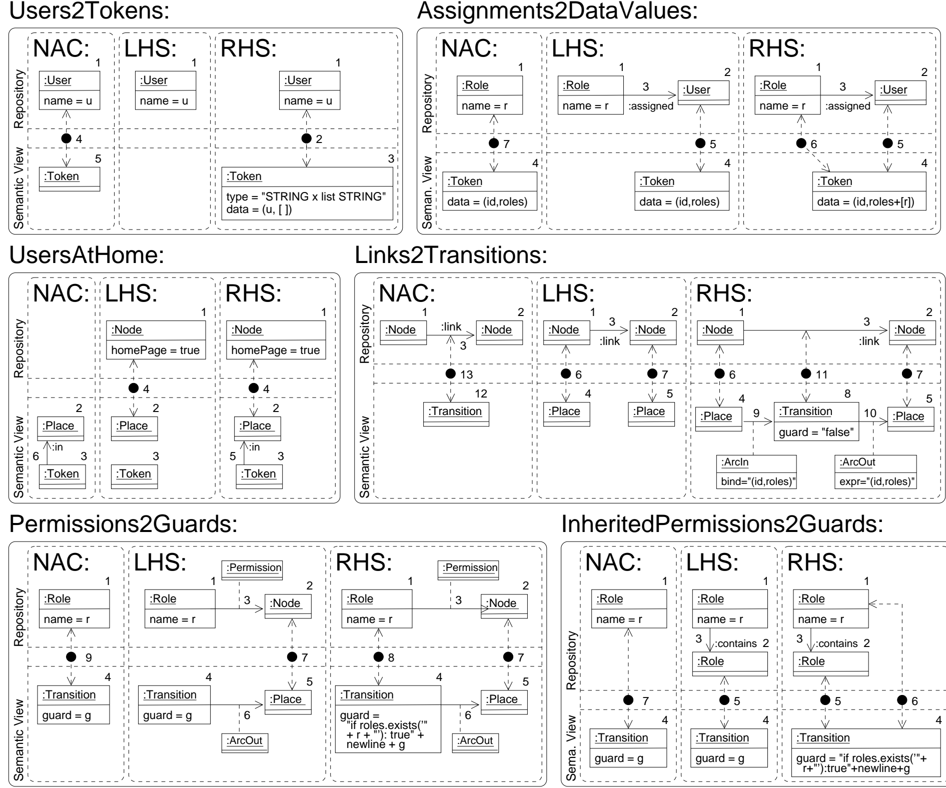



**Fig. 12.** Semantic View Definition by means of a Triple Graph Transformation System.

guard of the transition in order to allow the container access the place related to the node. Note how, this TGTS has the potential to be incremental by adding rules for creation, deletion and edition (as previously done for system views) for each element in the source model meta-model.

Fig. 1 shows the ATT-graph resulting from the application of this TGTS to the source model in the upper part. Thus, the lower model is its CPN semantic view. It is possible to analyse the CPN, for example, to check the navigation paths for a certain user or to find unreachable nodes for any user.

## 6 Related Work

TGTSs are a natural approach to handle views and model transformations. For example, [8] proposes TGGs for view creation. They work with views that can be considered model transformations, similar to our semantic views. They do not consider neither system nor derived views. Moreover, our formalization of triple rules [6] is more expressive as we allow attributes on nodes and edges, more flexible correspondence functions, application conditions and inheritance.

This work is also related to views in database systems [14]. These are virtual relations over a set of real data that are made visible to the user with the purpose of hiding information, or presenting such information in a more adequate way. Such views are defined as the result of a query (in a similar way as our derived views), and present problems for updating the real data as a result of a change in the view [10]. Our work is also related to the problem of consistency of replicated and fragmented data in distributed databases [14]. However, our concept of repository (i.e. we have a centralized control) and the model view controller approach we follow permits an easy solution to the consistency problem.

There are other approaches in the literature to express queries with graphs. For example, in VIATRA [1], queries on graphs can be expressed by generalized (recursive) graph patterns, which may contain a nesting of positive and negative patterns of arbitrary depth. Rensink [12] showed that this possibility is indeed equivalent in expressive power to first order logic. However, incremental transformations are not supported in VIATRA. Other possibility for queries is to use a textual notation, such as OCL [15]. We believe using a graphical approach makes the expression of complex structural patterns easier (in OCL it has to be coded by navigating between the relations), and may be more appropriate for non-computer scientists (patterns use the graphical notation of the given VL). On the other hand, OCL is much more expressive than our patterns (with only one level of nesting), and allows for example to express the absence of cycles. Graph patterns have been discussed extensively in the literature [3], especially its connection with application conditions for rules [7].

The QVT [11] specification includes a facility for queries (in addition to OCL), the *helper*, which allows combining blocks of expressions in a procedural way. Besides, it is also possible to define transformation rules to extract a derived view from a base model in the way we have presented here. Nonetheless, our approach is higher-level and declarative: starting from visual patterns, the rules that perform the transformation are automatically generated. By using QVT, the transformation to extract the view has to be coded by hand. As in our approach, QVT provides a mechanism (similar to the correspondence graph in triple graphs) for leaving traces (mappings) between the source and the target model, which allows a bidirectional update.

## 7 Conclusions and Future Work

In this paper we have proposed an approach for the uniform specification and handling of the different kind of views in MVVLs. The approach is based on meta-modelling for describing the syntax of the MVVL and its different diagram types. From the meta-models, triple rules are derived in order to build a unique model from the different system views that the user inputs, and to keep them consistent. Several alternative rules allow configuring the behaviour of the MVVL modelling environment. Derived and audience-oriented views are specified through graph query patterns. From these, a TGTS is generated that builds the derived view and keeps it consistent with the base model. Semantic views result from the

transformation of a base model into another formalism. Altogether, our approach makes emphasis on using visual, declarative techniques (meta-models, patterns), from which TGTSs are derived. However, for the case of semantic views, the TGTS has to be specified by the MVVL designer.

There is an ongoing implementation in the meta-modelling tool AToM$^3$ [2] [5]. Up to now, it is possible to define MVVLs and automatically generate the consistency triple rules for several behavioural patterns. Besides, it is also possible for the MVVL designer to define extra static semantics consistency rules as well as semantic views. It is up to future work to implement the graph query patterns. In addition to this, we are studying ways of improving the expressivity of the graph query patterns, and of DPO rules for model transformation.

**Acknowledgements:** This work has been sponsored by the Spanish Ministry of Science and Education, projects TSI2005-08225-C07-06 and TSI2004-03394. The authors would like to thank the referees for their useful comments.

## References

1. Balogh, A., Varró, D. 2006. *Advanced Model Transformation Language Constructs in the VIATRA2 Framework.* To appear in ACM SAC'06.
2. de Lara, J., Vangheluwe, H. 2002. *AToM$^3$: A Tool for Multi-Formalism Modelling and Meta-Modelling.* FASE'02, LNCS 2306, pp.: 174-188. Springer.
3. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G. 2006. *Fundamentals of Algebraic Graph Transformation.* Monographs in Theoretical Computer Science. Springer.
4. Ferraiolo, D., Cugini, J., Kuhn, D. R. 1995. *Role-Based Access Control (RBAC): Features and Motivations.* Computer Security Applications, pp.: 241-248. Springer.
5. Guerra, E., Díaz, P., de Lara, J. 2005. *A Formal Approach to the Generation of Visual Language Environments Supporting Multiple Views.* Proc. IEEE VL/HCC. pp.: 284-286.
6. Guerra, E., de Lara, J. 2006. *Attributed Typed Triple Graph Transformation with Inheritance in the Double Pushout Approach.* Tech. Rep. Universidad Carlos III. Available at `http://www.ii.uam.es/~jlara/investigacion/techRep_UC3M.pdf`
7. Heckel, R., Wagner, A. 1995. *Ensuring consistency of conditional graph rewriting - a constructive approach.* Proc. SEGRAGRA'95, ENTCS Vol 2.
8. Jakob, J., Schürr, A. 2006. *View Creation of Meta Models by Using Modified Triple Graph Grammars.* Proc. GT-VMT'06, to appear in ENTCS (Elsevier).
9. Jensen, K. 1992. *Coloured Petri Nets. Basic Concepts, analysis methods and practical use (vol. 1).* EATCS Monogr. on Theoretical Computer Science. Springer-Verlag.
10. Kozankiewicz, H., Subieta, K. 2004. *SBQL Views - Prototype of Updateable Views.* Proc. Advances in Databases and Information Systems (ADBIS).
11. QVT specification by OMG: `http://www.omg.org/docs/ptc/05-11-01.pdf`.
12. Rensink, A. 2004. *Representing First-Order Logic Using Graphs.* Proc. ICGT'04, LNCS 3256, pp.: 319-335. Springer.
13. Schürr, A. 1994. *Specification of Graph Translators with Triple Graph Grammars.* In LNCS 903, pp.: 151-163. Springer.
14. Silberschatz, A., Korth, H., Sudarshan, S. 2005. *Database System Concepts, $5^{th}$ Edition.* McGraw Hill.
15. Warmer, J., Kleppe, A. 2003. *The Object Constraint Language: Getting Your Models Ready for MDA, $2^{nd}$ Edition.* Pearson Education. Boston, MA.