Journal of Visual Languages & Computing 19.4 (2008): 429–455

DOI: http://dx.doi.org/10.1016/j.jvlc.2008.04.004

Enforced Generative Patterns for the Specification of the Syntax and Semantics of Visual Languages

Paolo Bottoni [a], Esther Guerra [b], Juan de Lara [c,*]

[a] *University of Rome "Sapienza", Comp. Science Dep., Rome (Italy)*

[b] *Universidad Carlos III, Comp. Science Dep., Madrid (Spain)*

[c] *Universidad Autónoma, Polytechnic School, Madrid (Spain)*

Abstract

We present the new notion of enforced generative pattern, a structure that declares positive or negative conditions that must be satisfied by a model. Patterns are applied to transformation rules resulting in new rules that modify models according to the pattern specification. In the case of a negative pattern, an application condition is added to the rule. In the case of a positive one, the rule is modified to consider additional context in its left-hand side and to increase its effects. We have defined these patterns in an abstract setting, which enables their instantiation for different structures, like graphs, triple graphs and graph transformation rules.

We apply the previous concepts to the specification of the syntax and semantics of visual languages. In particular, we show instantiations for: (i) graphs, with applications at the syntactical level; (ii) triple-graphs, for the coordination of syntax and static semantics; and (iii) rules, for the incremental construction of execution rules. We present some examples that illustrate the usefulness of the combination of these three instantiations. In particular, we show the specification of environments for visual languages with token-holder semantics, discrete-event semantics and communication semantics.

Key words: Visual Languages, Graph Transformation, Triple Graph Grammars, Graph Constraints, Patterns, Meta-Modelling, Specification of Semantics

* Corresponding author.
Email addresses: `bottoni@di.uniroma1.it` (Paolo Bottoni), `eguerra@inf.uc3m.es` (Esther Guerra), `jdelara@uam.es` (Juan de Lara).

1 Introduction

The design of Domain Specific Visual Languages (DSVLs) implies the definition of their syntax, usually derived from the notations in use in the domain community, as well as of their static and dynamic semantics [11]. The former includes the interpretation of the syntactic concepts in terms of the relevant semantic elements (e.g. the fact that a circle at the syntactic level is interpreted as a holder for tokens). Dynamic semantics can be given in an operational way, by specifying how the different semantic elements evolve with respect to time. Different approaches can be used, with varying levels of integration and incrementality between construction of syntactic sentences and interpretation in terms of abstract syntax or static semantics.

Meta-models are gaining popularity as a way to define the characteristics of both syntax and semantics, so that designing a new language involves the specialization of meta-classes and their relations [6]. Using meta-models, elements of concrete and abstract syntaxes are defined as instances of abstract concepts and constraints on their possible relations are given. The same mechanisms are used to define the semantic roles that elements can play [10]. Designers of new languages can thus map different concrete syntaxes to a common abstract one, given as a meta-model, and reuse significant parts of a language definition, in particular through inheritance [6,9,17]. However, the specification of the language semantic aspects – referring to the processes defined and simulated via the visual language – and of their connection to the syntactic ones is in many cases carried out by hand and from scratch, if no predefined relation is established between syntactic objects and semantic roles.

In previous works, we introduced the notion of *semantic variety*, expressed through a meta-model identifying the dynamic roles played by syntactic elements [5], and proposed *triple patterns* as a mechanism to generate triple graph operational rules (see [8,22,32]) coupling syntactic and semantic roles simply starting from the definition of syntactic rules [10]. Moreover, we proposed *action patterns* as a mechanism to generate rewriting rules expressing the execution semantics [4]. The application of these patterns allows the incremental generation of both the static and the execution semantics, in terms of transformations occurring with respect to designated elements of the visual sentence under construction. In particular, basic patterns have been defined for the *token-holder* transition semantic variety, in which discrete transformations occur by removing and adding tokens from and to holders representing some condition. Typical examples of languages presenting (discrete) semantics of this form are Finite State Automata, the different types of Petri nets, or workflow languages, but also languages based on positioning of elements in a grid, such as Agentsheets [28], or those describing chessboard games.

In this paper, we generalize these previous results by introducing a general notion of *enforced generative pattern* (EG-pattern) as a way to specify constraints in a declarative way. In particular, we set our study in the context of adhesive High-Level Replacement categories [12], and define a pattern as an object in such a category. When a pattern is applied to a rule based on morphisms in the same category, the rule is modified to ensure the production of an object conformant to the pattern. We present a general algorithm for pattern application entirely expressed in categorical terms. The algorithm can act on the different components of a rule, extending its effects and/or enriching its context, to make its effect conform to the pattern. Then, we show that both triple patterns and action patterns are special cases of EG-patterns. In particular, triple patterns are an instantiation for the category of triple graphs [17], while action patterns are an instantiation for the category of Double Pushout rules [12]. Thus, the application of these two different kinds of patterns to rules is based on a specialization of the algorithms given for EG-patterns.

We illustrate the potential of this approach by presenting its application to a number of DSVLs for specifying discrete systems and communication structures. Moreover, we introduce some extensions to EG-patterns, providing a rich catalog of tools to simplify the task of designing new visual languages or syntax-directed integrated environments.

Paper Organization. Section 2 introduces related research. Section 3 presents basic background on graph transformation and meta-modelling for defining syntax and semantic roles. In Section 4, we motivate the approach by presenting some situations which could benefit from the pattern concept. Section 5 introduces additional background, concerning triple graph grammars and meta-rules. Section 6 presents the algorithms for the application of positive enforced generative patterns. Section 7 shows an instantiation of patterns for triple graphs and its application to the coordination of the syntax and the static semantics of DSVLs. Section 8 instantiates patterns for rules (i.e. action patterns) and uses them for the specification of execution semantics of DSVLs. Section 9 gives some examples for languages with token-holder, discrete-event, and communication semantics. Section 10 introduces advanced pattern concepts, while Section 11 ends with conclusions and prospects for future work.

2 Related Work

The proposal of EG-pattern relates to different approaches to the expression of constraints on the effect of rules, as well as to specific mechanisms for their instantiation for triple graph grammars or execution rules. As a consequence, we touch on work concerning all these aspects.

Patterns expressing graph constraints were proposed in [19], and an algorithm was given to translate them into rule post-conditions, and then to pre-conditions. The algorithm does not affect the rule actions (i.e., the elements that the rule adds or deletes), but complements a rule with pre-conditions ensuring that, if the rule is applied, the resulting graph is consistent with the original graph constraint. Differently from [19], we are interested in modifying the rule actions, so that the produced graph conforms to the pattern. In a sense, the work in [19] can be seen as a particular case of the algorithms we provide. This is so as we give a whole spectrum of possibilities for applying the patterns, balancing how much the rule pre-conditions are modified with respect to the changes in the rule actions (the less the pre-conditions are changed, the more changes have to be done to the actions and vice-versa). The situation where the rule context is maximally extended and the rule actions are not changed corresponds to the ideas in [19]. Moreover, our patterns are expressed in categorical terms, so that they can be instantiated by different categories, like graphs, triple graphs or graph transformation rules.

The approach presented here also relates to the notion of manipulation of rules by means of rules, as proposed for graph transformation [25] and subsequently extended to High-Level Replacement Systems [26,29]. This is based on the definition of rule refinement through *rule* [29] and *subrule* [25] *morphisms*. Applications are found in termination analysis [7], as well as in management of security policies [21]. In [29], an algebra of rules is defined based on rule morphisms, including operators for rule composition. Multiple matches for a rule into another one give rise to different versions of the transformed rule. In this paper, we explore situations in which either several rules or a single one can be derived from a specific rule through pattern application, but also discuss the possibility of iterating the process to arrive at progressively refined rules. Our approach deals with *local*, as opposed to *global*, transformations [26]; in particular, it can also be used to generate rules by specialization, analogy or inheritance, but in a different way from [26].

The specialization of enforced generative patterns to the synchronization of syntactic and semantic rules relies on the notion of *Triple Graph Grammars* (TGGs, see [32]) and provides an efficient way to obtain TGG operational rules, whenever a grammar for one of the graphs already exists. In this scenario, patterns need not specify which elements should be created and which should already exist in one of the graphs – as is needed for the traditional specification of declarative TGG rules [32] – as this is expressed in the normal rule to which the pattern is applied. Thus, patterns may be used in several ways, providing a more flexible and declarative usage.

Abstract patterns [4,10] (in their different instantiations) exploit meta-models to express in a compact form a number of concrete patterns where an instance of a class is replaced by an instance of some given concrete subclass. In general,

the notion of abstract pattern can be employed in any situation where a meta-model is available to characterize objects in the considered category. Note that this is unrelated to the use of the term "abstract pattern" by Pagel and Winter [24], who propose a meta-pattern to describe object-oriented design patterns, dealing with pattern instantiation but not pattern enforcement.

In the field of TGGs, approaches based on meta-models already consider inheritance (see [8,22]). We add the possibility of applying abstract patterns to rules that can themselves be abstract, and to generate operational TGG rules which discriminate types by using negative application conditions. A formalization of TGGs with inheritance can be found in [17]. The instantiation of EG-patterns to triple graphs can generate operational TGG rules for the scenarios in [22]. We use triple graphs exploiting morphisms from the correspondence graph to the other two graphs, whereas in Fujaba a correspondence node can be related to several nodes in either graph, to express many-to-many relationships [20]. Baar uses TGGs to connect concrete and abstract syntaxes of DSVLs, allowing the static verification of their conformity [1]. However, his proposal is related to the structure of the visual sentence, and not to its operational interpretation. Moreover, it does not exploit inheritance, and requires the presence of display managers relating abstract and concrete syntaxes.

In [16], Göttler proposes meta-rules to modify syntactic or semantic standard rules, describing a programming language as a triple formed by a syntax, a semantics, and a function ϕ to build the semantic model from the syntactic one. In our case, meta-rules are associated with and triggered by syntactic editing rules. Moreover, they are automatically generated from action patterns.

In Ermel and Bardohl's approach to animation, execution semantics is given in terms of transformations of configurations of the graph defining the process state, and analogous rules transform an associated visualization [13]. Rule morphisms then synchronize rule application in the process and visualization domains. Our approach could be applied to visualization by considering the relation between static semantics and syntactic sentences (see [10]).

In [3], static semantics is incrementally built via meta-rules defining the correspondence between the elements of the diagram notation and those of the semantic domain, represented by High-Level Timed Petri Nets. Notation families are also introduced, to model commonalities in notations with slight differences in their interpretation. Our action patterns are able to support the definition of different interpretations on the same notation and the same static semantics. Moreover, our notion of semantic variety also encompasses different notations, sharing a similar structure for their interpretation.

In [33], amalgamation is exploited to generate execution rules for graphs describing static semantics. A specialized global execution rule is generated by

considering all possible simultaneous matches for a set of rules, once the complete host graph has been produced. Thus, the generation of parallel rules is not incremental, but needs to consider the whole graph and requires the identification of the effects on the interfaces between rules. For action patterns, instead, different matches independently contribute to the generation of a meta-rule. Hence, by generating specific execution rules for each transition, we overcome some limitations of [33], in which, for example, checking in a Petri net whether all pre-conditions for firing a transition are satisfied is solved by specific Double Pushout idioms, such as rewriting the transition itself. This exploits the dangling edge condition (not present in other rewriting approaches) if some place does not have enough tokens (hence, not producing a match for the sub-rule). Our framework is not tied in principle to any specific rewriting approach and provides more concise specifications.

Patterns of execution have been studied in the modelling of workflow processes and a semantics for them has been given by Coloured Petri Nets [34]. As these may be expressed in terms of action patterns, the definition of a pattern language for workflows could benefit from the approach presented here.

3 Formal Background I

3.1 Introduction to Graph Transformation

Graph transformation [12,30] is a visual, formal, and declarative means to express graph manipulations, by defining a set of rules and an initial graph. Rules have left and right hand sides (LHS and RHS), each containing graphs. When applying a rule to a *host* graph G, a match morphism must be found between LHS and G. Then, the elements not preserved by the rule (roughly $LHS - (LHS \cap RHS)$) are deleted in G, and the new elements (roughly $RHS - (LHS \cap RHS)$) are added. This step is called direct derivation. The defined language is the set of all possible graphs obtained by iterating direct derivations starting from the initial graph.

The left of Fig. 1 shows the "move" rule and a direct derivation, using an abstract syntax representation similar to UML object diagrams. The rule is concerned with the simulation of an automaton-like visual language. It moves a `current` pointer between two states as effect of the execution of a transition. It is applied to a graph G, using a match m and yielding graph H. The picture shows the mapping for nodes, through equality of identifiers; a similar mapping for edges is also part of the match. The right of Fig. 1 shows the same rule in a compact notation that will be used throughout the paper. The elements added by the rule are enclosed in coloured regions, marked as "{new}". Similarly,

the deleted elements are enclosed in regions marked as "{del}".

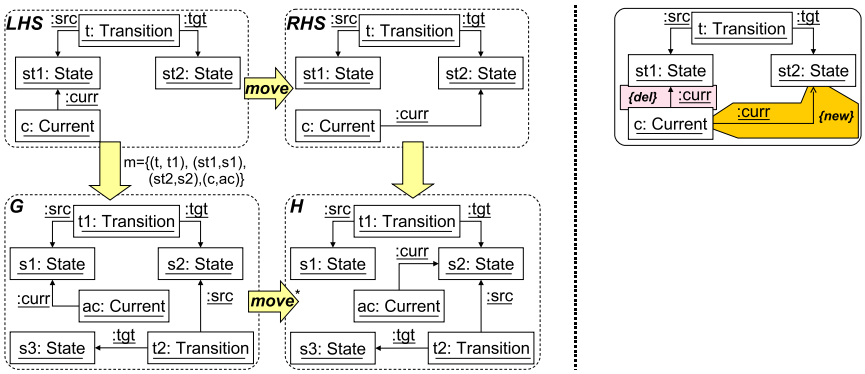

Fig. 1. Direct Derivation Example (left). Compact Notation for Rules (right).

One of the most popular formalizations of graph transformation is based on category theory and is called Double Pushout (DPO) [12]. In this approach, a direct derivation is modelled in two steps. First, elements are removed from the host graph according to the rule specification, and then the new elements are added. For this purpose, a rule is made of three component graphs: the left and right hand sides (L and R), and the interface graph K, which contains the elements preserved by the rule application. Two injective morphisms $l\colon K \to L$ and $r\colon K \to R$ model the embedding of K in L and R. The left of Fig. 2 shows a DPO direct derivation diagram. Square (1) is a pushout (i.e. G is the union of L and D through their common elements in K) that performs the deletion, while pushout (2) adds the new elements.

$$\begin{tikzcd} L \arrow[d, "m"'] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] \arrow[dl, phantom, "(1)"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "m^*"] \\ G & D \arrow[l, "f"'] \arrow[r, "g"] & H \end{tikzcd} \qquad \begin{tikzcd} Y_{ij} \arrow[d, "o_{ij}"'] & X_i \arrow[l, "y_{ij}"'] \arrow[r, "x_i"] \arrow[d, "n_i"] \arrow[dl, phantom, "="] \arrow[dr, phantom, "="] & L \arrow[l, "l"'] \arrow[d, "m"] \arrow[dl, phantom, "(1)"] & K \arrow[r, "r"] \arrow[d, "k"] \arrow[dl, phantom, "(2)"] & R \arrow[d, "m^*"] \\ & & G & D \arrow[l, "f"'] \arrow[r, "g"] & H \end{tikzcd}$$

Fig. 2. DPO Direct Derivation Diagram (left). Derivation for Rule with NAC (right).

Fig. 3 shows a DPO direct derivation example using the same rule and host graph as Fig. 1. The left square deletes the `curr` edge from the `current` node to state `s1`, while the right square adds an edge from the `current` node to `s2`. The figure also shows the fact that rules may have parameters, modelled by graph M and injective morphism $m_l\colon M \to L$. Parameters are used to initialize the match to which the rule is applied. In the example, the parameter initialization enforces the application of the rule to transition `t1`, should there be more than one option. Thus, the rule can only be applied at a match m if $m \circ m_l = m_M$. Throughout the paper, we prefer the compact notation for showing example rules. However, we will use the DPO notation of Fig. 3 for the theoretical presentation of the algorithms and corresponding examples.

Sometimes, rules are equipped with application conditions, expressing additional conditions that the match should satisfy to make the rule applicable [12].

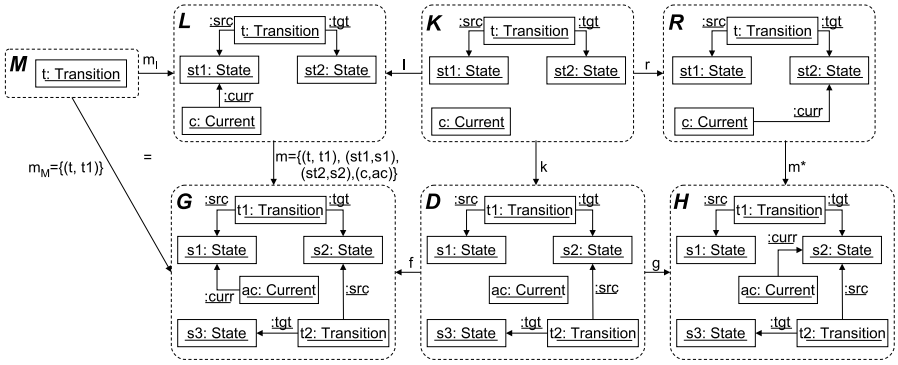

Fig. 3. DPO Direct Derivation Example.

Particularly useful are *negative application conditions* (NAC), given by additional graphs N_i related to L by morphisms $n_i\colon L \to N_i$. The rule is applicable if, for each i, there is no morphism $q_i\colon N_i \to G$ commuting with m (i.e. such that $q_i \circ n_i = m$). We also use more general conditions of the form $\{x_i\colon L \to X_i, \{y_{ij}\colon X_i \to Y_{ij}\}_{j\in J_i}\}_{i\in I}$, which are satisfied by a match $m\colon L \to G$ if, for each $n_i\colon X_i \to G$ such that $n_i \circ x_i = m$, there exists some $o_{ij}\colon Y_{ij} \to G$ such that $o_{ij} \circ y_{ij} = n_i$ (see the right of Fig. 2). Note that a NAC is an application condition where $J_i = \emptyset$.

The previous examples use typed graphs as the underlying data structure. Formally, a *type graph* is a construct $TG = (N_T, E_T, s^T, t^T)$ with N_T and E_T sets of node and edge types, respectively. $s^T\colon E_T \to N_T$ and $t^T\colon E_T \to N_T$ define the source and target node types for each edge type. A typed graph on TG is a graph $G = (N, E, s, t)$ equipped with a graph morphism $type\colon G \to TG$, composed of two functions $type_N\colon N \to N_T$ and $type_E\colon E \to E_T$ preserving the s^T and t^T functions. In this paper, we use type graphs with node type inheritance, defined by a pair $TGI = (TG, I)$, where $I = (N_I, E_I, s^I, t^I)$ and $N_I = N_T$. Hence, I has the same nodes as TG, but its edges are the inheritance relations. The inheritance *clan* of a node n is the set of all its children nodes (including n itself): $clan(n) = \{n' \in N_I | \exists \text{ path } n' \to^* n \text{ in } I\} \subseteq N_I$.

For different applications, other structures may be more appropriate, like attributed (typed) graphs, triple graphs or P/T nets. These considerations have led to generalizing the graph transformation theory to higher-level structures: (weak) adhesive High-Level Replacement categories, short (w)aHLR [12], so that DPO rewriting can be used with objects in any (w)aHLR category.

3.2 Meta-Models in the Definition of Syntax and Static Semantics

We adopt a meta-modelling approach for the definition of syntax and static semantics for diagrammatic languages, based on the classes of Fig. 4, which shows two meta-models related through a *correspondence meta-model*. This

structure is called *meta-model triple* [17] and we use it to describe two related languages in a modular way (in this case, one for expressing concrete syntax, the other for static semantics). The correspondence meta-model is used to relate concepts in both languages, therefore its nodes have morphisms to nodes or edges in the other two meta-models. In the meta-model for concrete syntax shown in the lower part of Fig. 4, semantic relations are expressed via *spatial relations* between *identifiable elements*. Different specialisations of `IdentifiableElement` and `SpatialRelation` define different *families* of visual languages [6], such as the connection- and containment-based ones. Identified elements are put in correspondence with semantic roles, as defined by the semantic variety to which the visual language belongs.

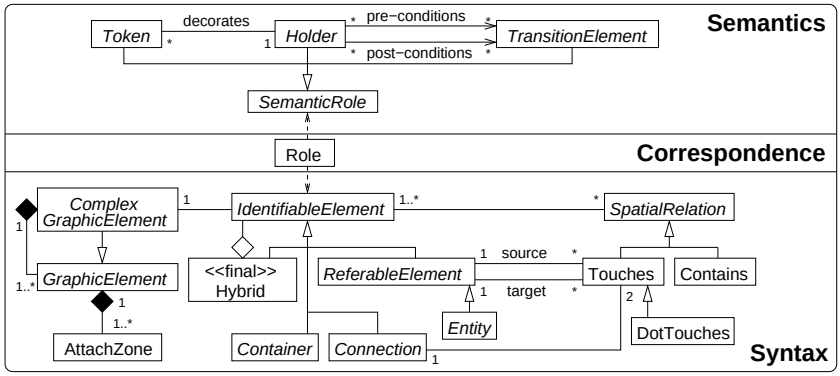

Fig. 4. Meta-model Triple for Syntax and Semantics of Visual Languages.

The upper part of Fig. 4 presents the fundamental classes for the *transition* semantic variety. In general, this relies on some notion of *configuration* of a system, which is significantly changed by the firing of the transition. Hence, the transition variety collects uses of visual languages to describe transformation processes in which a diagram depicts an instantaneous configuration, evolving under some well-defined law. The possible evolutions at each step can be statically derived from the form of the diagram (e.g. transitions in Petri nets), or described externally (e.g. by grammar rules). Internal descriptions of the admissible transformations rely on the presence of identifiable elements which directly represent `Transitions`, with which `Holder` elements are associated as either `pre-` or `post-conditions`. Examples of such direct representations are arrows and nodes in finite state machines, or boxes and circles in Petri nets. Associations between holders and transitions allow the specification of the static semantics associated with a diagram, while its execution semantics is defined by some external interpreter, and results into deleting or creating associations between `Token` and `Holder` elements. In particular, this can be given through rules of type *before-after*, based on the differences in the way `Tokens` decorate `Holders`. For example, in grid-based languages, holders are grid cells and tokens are symbolic representations of the domain elements. An execution semantics in terms of before-after rules can also be imparted on transition-based languages by specifying, for each transition, the movement of

tokens from pre-condition to post-condition holders.

The concrete and semantic roles in a visual language are defined by refining the previous meta-model triple. For example, the left of Fig. 5 shows elements in the concrete syntax and semantic roles for Petri nets. The significant spatial relations are refined (via a creation graph grammar, which specifies how the different elements are connected) to be the `Touches` relation between instances of `ArcPT` (`ArcTP`) and a source `Place` (`Transition`) or a target `Transition` (`Place`), and the `Contains` relation between `Place`s and `Tokens`. A `Place` can play both the role of an `Entity`, with respect to the arcs referring to it, and of a `Container`, with respect to the `Tokens` it holds. The right of Fig. 5 shows a triple graph example conformant to the meta-model triple to its left.

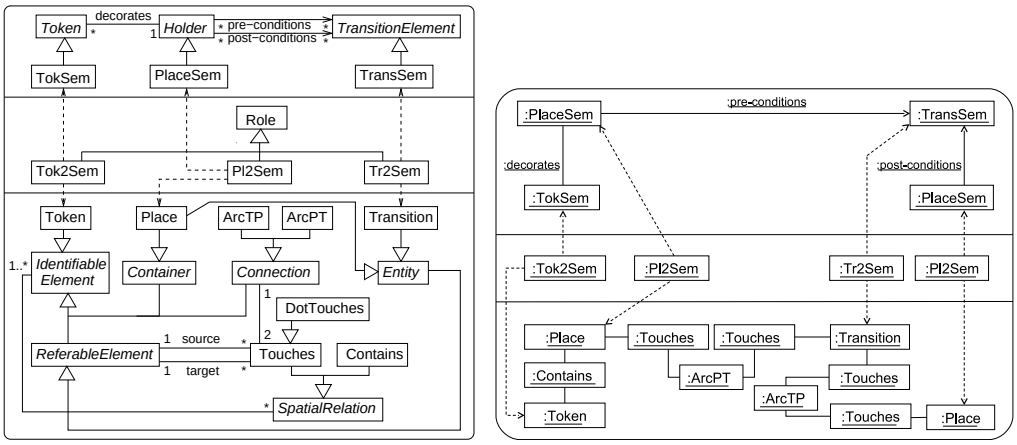

Fig. 5. Meta-Model Triple for the Definition of Petri Nets (left). Triple Graph with Concrete and Semantic Roles for a Petri Net Model (right).

According to the adopted syntax, the representation of the system dynamics can be directly supported by some canonical animation, in which instances of the `Token` abstract class (in the semantics model) may appear or disappear, or move from one instance of a `Holder` to another. It is to be noted that the meta-model definition of the static and execution semantics allows the adoption of different syntactic representations for the same semantics, provided that some equivalence can be established between the two. For example, a Petri net can be represented by replacing transition boxes with hyperedges.

4 From Syntax to Execution through Static Semantics

This section presents a motivating example, introduces the mechanisms exploited in the paper and explains their benefits. Suppose you are planning the development of an integrated environment for designing and executing Petri

nets, providing a syntax-directed editor in which a designer can directly manipulate places and transitions. The environment maps concrete elements onto an abstract syntax, acting as a repository for the constructed net, in correspondence with a static semantics, seeing the net as a collection of transitions with pre- and post-conditions. Advanced editing commands allow one to create transitions with associated pre- and post-condition places, introduce conflicts between two existing transitions, or add cascading transitions to existing ones.

The abstract syntax exploits elements of type `Place`, `Transition`, `ArcPT`, and `ArcTP`, while the static semantics assigns to `PlaceSem` elements, representative of places, the roles of pre- or post-conditions with respect to `TransSem` elements, denoting transitions (see the meta-model triple in Fig. 5).

Fig. 6 shows the concrete and abstract syntax rules for creating a transition, with associated source and target places, and updating the static semantics.

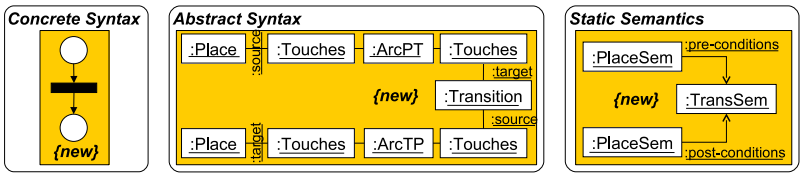

Fig. 6. Rules to Update the Concrete Syntax, the Abstract Syntax and the Static Semantics when Creating a Transition.

The execution mechanism we propose associates each transition with a specialized rewriting rule that checks that all pre-condition `PlaceSem` elements for the corresponding `TransSem` element are decorated with `Token` elements, removes them, and decorates all its post-conditions with new tokens. Thus, each rewriting rule can only be executed at a particular transition. We rely on a mechanism that initializes the match of the execution rule with the particular transition node, restricting its application. Whenever such match initialization is important, we express it through parameters, as shown in Fig. 3.

A different approach could define a set of graph transformation rules, detecting when a Petri net transition is enabled and firing it. This has several disadvantages: (i) several rule executions are needed to simulate the actual firing of a Petri net transition, so that execution is less efficient and analysis harder, (ii) auxiliary elements would be needed to detect that the transition pre-places have enough tokens, or to add tokens in post-places. As a result, fewer rules could be needed, but at the price of modifying the Petri net meta-model. Moreover, generating customized rules allows their static analysis, independently of the host graph. This in fact is a standard procedure when representing Petri nets with graph transformation rules [23]. Finally, the mechanism we propose in this paper saves the designer from building such execution rules, while the fact that a rule is generated for each transition is transparent to the end user, and can lead to faster pattern matching.

Fig. 7 presents a meta-rule (i.e. a rule transforming a rule) that initializes the execution rule associated with the transition created by the rules in Fig. 6. Note how the meta-rule updates rules that refer to specific elements created by the static semantics rule, against which they should only be instantiated. This means that the execution rule can be applied only to the `TransSem` element created by the static semantics rule of Fig. 6. This is ensured by parameter passing: the meta-rule initializes the execution rule with one parameter, which is used to pass the `TransSem` node created by the static semantics rule.

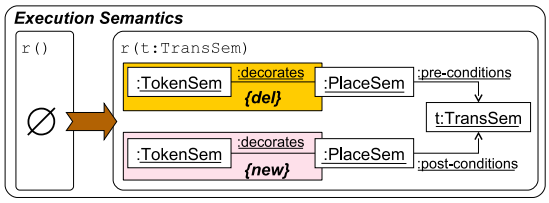

Fig. 7. Rule to Initialize the Execution Semantics upon Generation of a Transition.

Fig. 8 shows the concrete and abstract syntax rules for inserting a conflict, as well as the update of the static semantics with the insertion of a `PlaceSem` related by the `pre-conditions` association with each of the two `TransSem`.

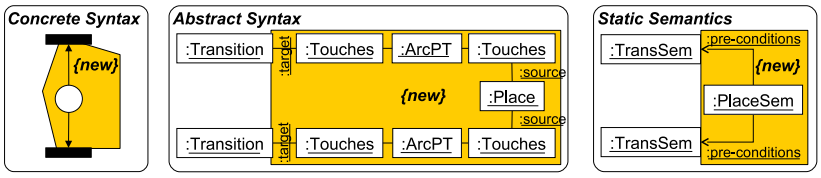

Fig. 8. Rules to Update the Concrete Syntax, Abstract Syntax and Static Semantics for Insertion of a Conflict.

Fig. 9 describes the updating of the execution rules associated with the two transitions. The `TransSem` nodes "t1" and "t2" have to be the same as those in Fig. 8 (in the upper and lower part of the rule for static semantics) so they will be passed as parameters. As the same `PlaceSem` element appears as a precondition in both rules, the rules are in conflict, which will be managed by the execution mechanism, for example along the lines of [18].

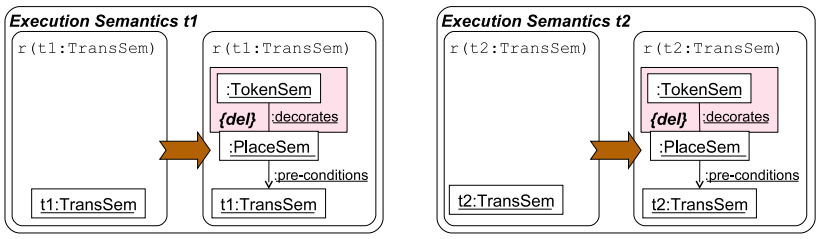

Fig. 9. Rules to Update the Execution Semantics for the First (left) and Second (right) Transitions in Conflict.

Fig. 10 describes the concrete, abstract syntax and the static semantics for the construction of a chain of cascading transitions, while Fig. 11 refers to

the updating of the execution semantics. In this case, it is important to note that while the upper transition is already present, so that the corresponding rule has to be updated, the lower one initializes the rule associated with the transition created by the syntactic rule.

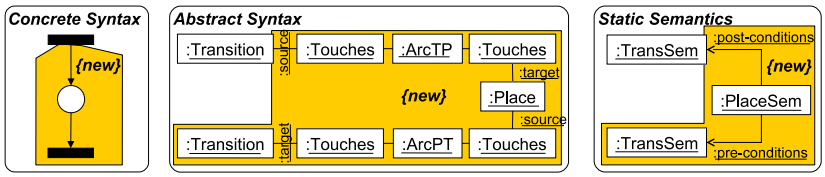

Fig. 10. Rules to Update the Concrete Syntax, Abstract Syntax and Static Semantics when Constructing a Transition Chain.

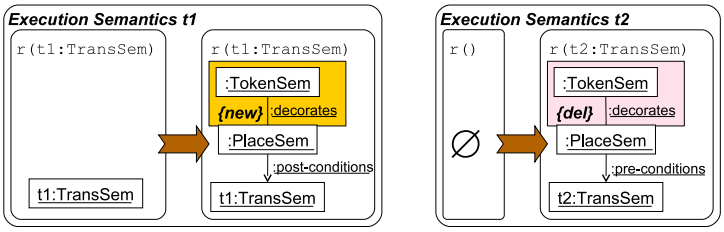

Fig. 11. Rule to Update the Execution Semantics for the Upper (left) and Lower (right) Transition in the Chain.

It is easy to observe the existence of a common pattern both in the construction of the static semantics given the concrete syntax (or equivalently the abstract one) and in the definition of the execution semantics. In particular, in the static semantics, a `pre` (`post`)`-conditions` relation is constructed for each combination of `ArcPT` and `source` and `target` associations in the abstract syntax. In the execution semantics, a `Token` is removed from each *pre* `PlaceSem` and added to each *post* `PlaceSem`.

Hence, the syntactic rule should suffice to determine the updating of both static and execution semantics, according to the conventions illustrated above, without forcing the designer of the editing system to provide the execution and abstract syntax rules, which could be automatically generated. A designer can thus create arbitrarily complex syntactic rules, relying on automatic generation of the abstract syntax and execution rules, avoiding their tedious and error-prone manual generation. In the rest of the paper, we show how EG-patterns can support both the definition of the execution semantics (by their instantiation into action patterns) and the coordination of syntax and static semantics models for DSVLs (by their instantiation into triple patterns). First, we introduce additional background concepts regarding TGGs and meta-rules.

5 Formal Background II

5.1 Triple Graph Grammars

Graph transformation is a natural way to specify in-place transformations, like model animation and refactoring. However, in model-to-model transformation a source model conforming to a meta-model is transformed into a target model conforming to a different one. Thus, for this kind of transformations, it is preferable to have a means to cleanly separate source and target models (as well as their meta-models), so as to establish mappings between them.

Triple Graph Grammars (TGGs) [32] were invented by Andy Schürr as a means to translate and synchronize two different graphs (called *source* and *target* graphs) related by a *correspondence* graph. The nodes in the correspondence graph have morphisms to nodes in the source and target graphs. This structure, an example of which was given on the right of Fig. 5, is called *triple graph* and is represented as $G = (G_s \overset{cs}{\leftarrow} G_c \overset{ct}{\rightarrow} G_t)$. We allow triple graphs to be typed over a meta-model triple (see the left of Fig. 5). In previous research [17], we showed that attributed typed triple graphs form an aHLR category and can thus be manipulated through DPO rules where the L, K and R components are triple graphs.

TGG rules are useful for model-to-model transformation, allowing incrementality and a certain degree of bi-directionality. Starting from high-level, *declarative* TGG rules (like a *creation* grammar for a triple graph language), one derives *operational* rules with different purposes: source-to-target or target-to-source translation, incremental updates or model synchronization [22].

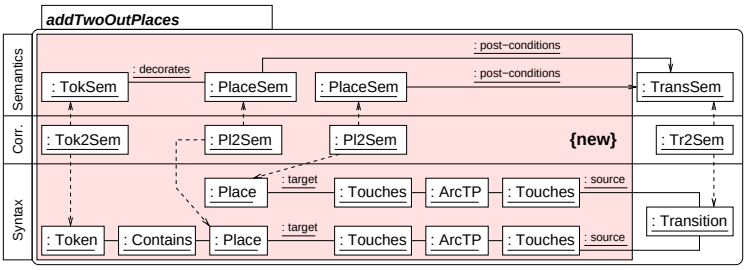

Fig. 12. TGG Rule Example.

Fig. 12 shows an example TGG rule that models the addition of two output places (one of which containing a token) to a transition, both in the concrete syntax and in the semantics model. From this synchronous rule, the algorithms in [32] generate lower level rules, in this case to update the semantic model according to the modification in the syntax. However, this solution is not optimal if we are designing syntax-editing rules for a modelling environment, as in Section 4. In this case, one would like to design rules taking into

consideration the concrete syntax only; and then have some mechanisms to propagate such changes. Sometimes, a synchronous modification of the semantic model is preferred. Moreover, a synchronous mechanism becomes essential when performing an execution at the semantic level, and we want to observe the animation at the concrete syntax level. Again, instead of building TGG rules by hand, a designer can automatically obtain them starting from the execution rules and mechanisms relating syntax and semantics. In Section 7, we give an instantiation of enforced generative patterns to triple graphs performing exactly this task [10] and minimizing the effort needed for specifying consistency mechanisms between syntax and semantics.

5.2 Meta-Rules

In order to specify the execution semantics of a visual language, we rely on the identification of *active elements*, such as transitions in Petri nets or state automata, for which execution rules are created. Each execution rule is associated with one particular active element and models its dynamic semantics. *Editing rules* create and connect elements of the language and are used to build the model. Fig. 13 shows the syntactic rule *addPlaces*, and the corresponding semantic one, to add an incoming and an outgoing place to a transition.

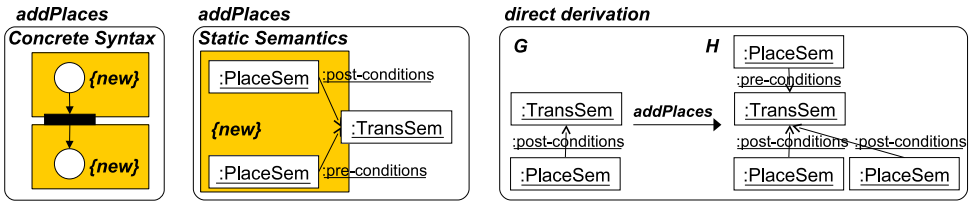

Fig. 13. An Editing Rule Example in Concrete Syntax (left). Derived Rule for the Static Semantics Model (center). A Direct Derivation (right).

In our approach, editing rules are paired with one or more *meta-rules* to update the associated execution rule for each transition element considered by the editing rule. A meta-rule is invoked each time the corresponding syntactic rule is applied. The meta-rule modifies the execution rule for the involved transition elements, in order to obtain a customized rule reflecting the exact context (exact number and identities of pre- and post-conditions) in which the transition can perform a transformation step. Hence, meta-rules are DPO rules modifying rules (i.e. each of the L, K and R components of a meta-rule is in turn a DPO rule). This is possible, as DPO rules can be shown to form an aHLR category. Briefly, if $\mathbf{C}$ is an aHLR category, then so is the functor category $DPO(\mathbf{C}) = [\cdot \leftarrow \cdot \rightarrow \cdot, \mathbf{C}]$. This category has as objects all possible functors from the scheme category $\cdot \leftarrow \cdot \rightarrow \cdot$ to $\mathbf{C}$ (where one such functor represents a DPO rule) and as arrows all natural transformations. Similarly, DPO rules with parameters also form an aHLR category, due to the fact that

$P\text{-}DPO(\mathbf{C}) = [\cdot \to \cdot \leftarrow \cdot \to \cdot, \mathbf{C}]$ is an aHLR category.

Suppose that the editing rule of Fig. 13 is applied to graph G to its right. After the application, the execution rule for transition t has to be updated so that it manages the deletion of a token from the newly created pre-condition place, and insertion of tokens in the resulting two post-condition places. Fig. 14 shows the process by which rule *Fire_t*, associated with the transition t, is transformed into *Fire_t"* to reflect the addition of the two new places to the transformed graph H. This is performed by the meta-rule shown in the upper part of Fig. 14, to be associated with the editing rule *addPlaces* of Fig. 13. In this case, the meta-rule does not modify the parameter of the execution rule (it is shown as a graph M in the meta-rule and textually in rule *Fire_t*). This parameter will be used to ensure that the execution rule can only be applied to the transition to which *addPlaces* was applied (see the remark in Section 4).

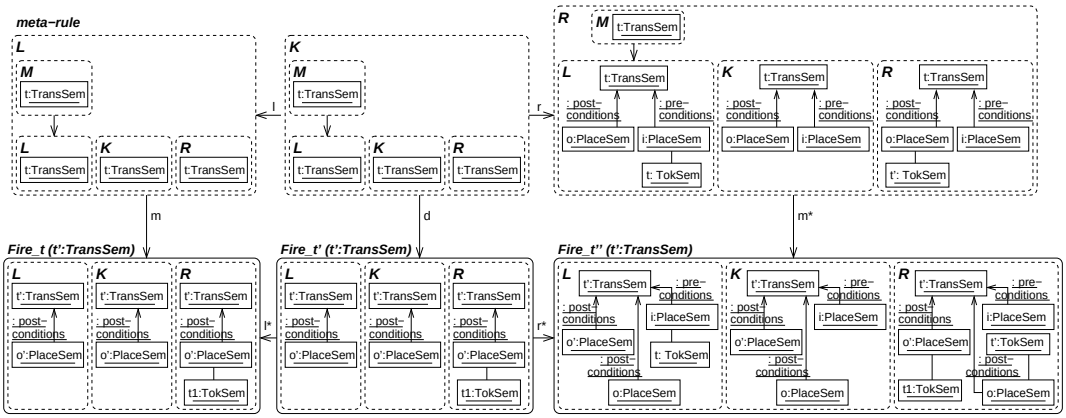

Fig. 14. Meta-Rule Derivation Example.

Thus, in order to deal with an incremental specification of a visual language execution semantics, we propose, in Section 8, another instantiation of EG-patterns for DPO rules (called action patterns). Given a set of action patterns, we create a meta-rule that updates the execution rule associated with each active element in the model modified by the editing rule, to accommodate the produced changes. The next section presents the notion of EG-pattern using category theory, to be instantiated later in a number of ways.

6 Positive Enforced Generative Patterns

This section presents the general notion of *enforced generative pattern* (short *EG-pattern*) and its application to DPO rules to yield modified rules producing objects conforming to a pattern. We give here the *simple* version of *EG-pattern*. More sophisticated ones, such as patterns with negative conditions and composite *"if-then"* patterns are left for Section 10.

A *simple positive EG-pattern* (short *SP-pattern*) is an object P in a (w)aHLR category. An object H satisfies P, written $H \models P$, if $\exists m\colon P \to H$ injective morphism. Intuitively, this means that at least one occurrence of P is found in H. For the moment, we take an *existential* view of pattern satisfaction. We sometimes use the term "H is conformant to (or consistent with) P".

The application of an *SP-pattern* P to a DPO rule $p : L \xleftarrow{l} K \xrightarrow{r} R$ yields a modified version of p that, when applied to an object G, produces an object H consistent with P. We consider non-deleting DPO rules, i.e. with $L = K$, hence $p : L \xrightarrow{r} R$. Constructions are similar for deleting (and non-creating) rules in which $K = R$ and roles of L and R are reversed. In order to produce a rule conforming to the pattern, we have two extreme solutions: extending only the RHS (i.e., the rule effects), or both the LHS and the RHS, but without creating any new element that the original rule did not create. There are also intermediate situations, where additional context is considered in the LHS and the rule effects are expanded. The three cases can be described with a single algorithm. However, for better understanding, we start with the two extreme situations and finish with the general algorithm.

Extending the rule effects. In order to make $p : L \xrightarrow{r} R$ enforce P, the first option is to extend p's effects. To this end, we build the diagram to the left of Fig. 15, where square (1) is a pushout. The new rule is then $p : L \xrightarrow{r'} R'$. In the diagram, intuitively, P' is an object that models a maximal intersection of R and P, and in general is not unique. The conditions that must be satisfied by a maximal intersection object P' are:

- P' should not be empty, i.e. it should not be an initial object in the given category. An empty intersection means that no partial, non-empty occurrence of the pattern P is found in R and thus the pattern does not have to be enforced by the rule.
- On the other extreme, P' should not be isomorphic to P, as otherwise the pattern is already enforced by the rule as it is.
- There is no span $(P'', p'\colon P'' \to P, pr'\colon P'' \to R)$ and injective morphism $p''\colon P' \to P''$ such that $pr' \circ p'' = pr$ and $p' \circ p'' = p$, with $P'' \not\cong P'$. This is needed in order to ensure that P' is maximal. A diagram showing this condition is depicted at the center of Fig. 15.

The previous procedure generates a rule that, when applied to an object G, produces an object H conformant to P. That is, it ensures that an occurrence of P exist in H. Sometimes, our aim is to complete all possible non-empty partial matches of the pattern in the rule's RHS, thus ensuring a *universal* satisfaction of the pattern (local to the rule's co-match). For this purpose, the procedure can be iterated for the new rule $p : L \xrightarrow{r'} R'$ selecting P'' as a maximal intersection of R' and P, different from P'. A typical use of the algorithm is to select in each iteration a maximal intersection object $P^{i'}$

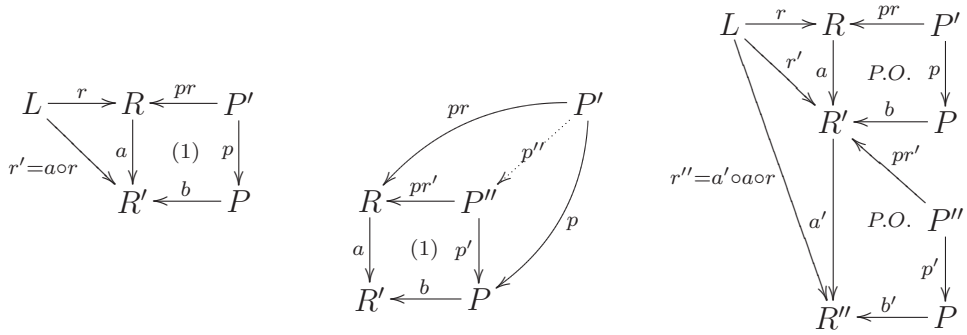

Fig. 15. Extending the Effects of a Rule According to *SP-pattern* P (left). Condition for Maximal Intersection Object P' (center). Iteration (right).

having a non-empty intersection with the R component of the original rule (and differing from the previous intersections). This ensures that the original rule is completed in all possible ways, and also guarantees termination of the iteration: as $P^{i'}$ has to be maximal and cannot grow more than P, there is only a finite number of possible matches onto the original R. The scheme of the iteration is shown on the right-hand side of Fig. 15.

Example. The left of Fig. 16 shows an example of the previous construction in the category of typed graphs. Node identities are used to indicate the morphisms. In the example, pattern P is applied to a rule that checks the existence of a node of type A and creates another node with type B connected to it. The pattern demands the existence of a structure made of a node of type A connected to two nodes of type B. A maximal intersection graph P' is calculated, and the pushout graph R' is obtained, so as to include in the rule's RHS the pattern elements which are not created by the rule. Thus, the pattern application ensures that the generated rule yields an object H satisfying the pattern P. The figure also shows that a smaller graph P'' (with just one node a) cannot be taken as maximal intersection object.

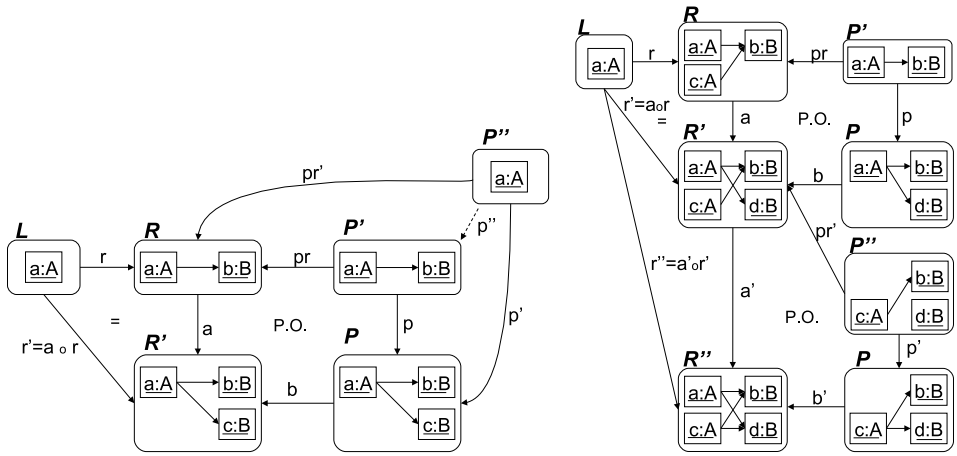

Fig. 16. Example of Extending the Effects of a Rule (left). Iteration Example (right).

The right of Fig. 16 shows an iteration until no valid maximal intersection object is found. Given one node of type A, the rule creates another one of the same type, both connected to a new node of type B. However, the pattern specifies that two Bs are necessary for each A. Thus, in a first step, the RHS is modified to connect the already existing A to a new node B. In a second step, this newly created B is reused and connected to the other A.

Note that the use of the procedure proposed in [21], calculating all the maximal intersection objects a-priori, would result in a bigger RHS, which is not desirable. In the example, the rule would create three Bs in total, instead of just two. With the iteration, we are reusing parts of the different RHSs, which results in the creation of the smallest object satisfying the pattern.

Extending the rule context. The second option is to maximally extend the rule's LHS to incorporate all the necessary context, so that the resulting rule does not have to create anything new to satisfy the pattern. That is, the rule effects are not modified. However, as we will see, this is not always possible.

To extend the rule, a maximal intersection object P' is chosen and the pushout object R' is obtained. Then, L is maximally completed to yield L'. The main idea is to partition P into P_L and P_R, where P_L is what will be added to L to yield L'. As we want to maximally extend L without modifying the rule effects, we take P_R as small as possible, thus we take it to be P'. As P_L has to be glued with L, we calculate the gluing object Q as the pullback[1] object of $r\colon L \to R$ and $pr\colon P' \to R$. In this way, Q is the intersection of pattern P_L with L. If P_L exists, it has to be the pushout complement of $q_P\colon Q \to P_R$ and $q\colon P_R \to P$ (as $P' = P_R$, we have $q = p$). If such P_L exists, then we calculate L' as the pushout of $q_L\colon Q \to L$ and $q_{P_L}\colon Q \to P_L$ to obtain L'. Note that $r'\colon L' \to R'$ uniquely exists due to the pushout universal property, as $a \circ r \circ q_L = b \circ c \circ q_{P_L}$. If no such P_L exists, then we cannot extend L, but we obtain a rule where only R is extended to R', and set $r' = a \circ r$.

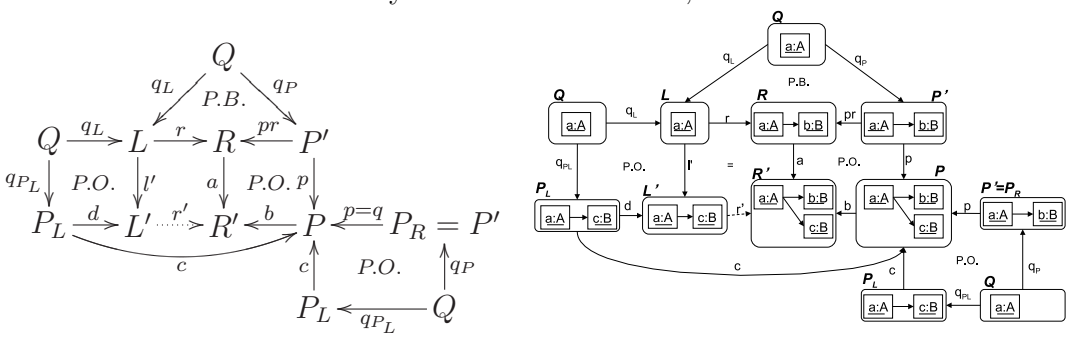

Fig. 17. Maximally Extending the Rule Context (left). Example (right).

As previously, the procedure can be iterated for each valid maximal intersec-

[1] Roughly, a pullback object is an intersection of two objects sharing a common context.

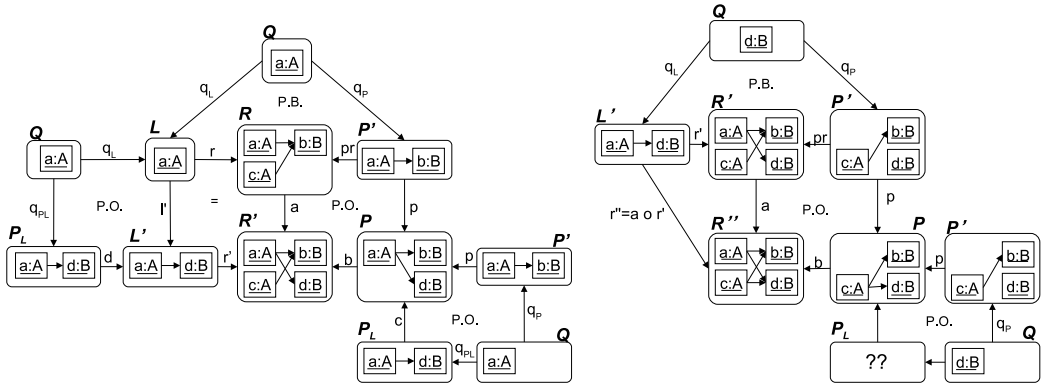

Fig. 18. Example: Maximally Extending the Context of a Rule, First Iteration (left). Second Iteration. Non-Existence of P_L (right).

tion object having a non-empty intersection with the original rule's RHS.

Example. The right of Fig. 17 shows a maximal extension of the LHS of a rule, according to pattern P (previously shown in Fig. 16). A maximal intersection P' is found, and then R' is calculated, extending R with one additional B connected to the A node. The pullback graph of pr and r is calculated, which contains only one node A. This graph is used to calculate the pushout complement (P_L) of p and q_P. As P_L exists in this case, we can extend L by considering the extra B connected to the existing A. Overall, the rule has been extended with additional context, but its effect is not enlarged. Note that the condition for the pushout complement for graphs to exist basically states that for any node $n \in P' - q_P(Q)$, all incident edges to its image $p(n)$ should have a preimage in P_R. This is similar to the dangling-edge condition for direct derivations. In the example, P_L would not exist if P had in addition an edge between the two B's. The existence of pushout complements for the case of (w)aHLR categories is discussed in Theorem 6.4 of [12].

The example shown in Fig. 18 presents an iteration, but in the second iterative step the pushout complement P_L does not exist, which means that we cannot extend the LHS twice [2]. Thus, the resulting rule is $p' \colon L' \to R''$

Intermediate situations. Now we consider situations where some part of the pattern is added to the context, and the rule effects are also expanded. We split pattern P into P_L and P_R, where, intuitively, P_L is added to the LHS, and when glued together (through some common subgraph S_P), $P_L +_{S_P} P_R$ yields P. Thus, the effects of the rule are enlarged according to P, and the LHS according to P_L. The situation is similar to the previous construction; however, P_L is now not required to be a maximal extension of L. The left of Fig. 19 shows the diagram for this construction. Again, morphism r' uniquely

[2] This can be regarded as a conflict between the pattern and the rule.

exists due to the pushout universal property, as $b \circ c \circ q_{P_L} = a \circ r \circ q_L$. P_L is glued with L through the pullback object Q of L and P'. We require an injective morphism $q_S \colon Q \to S_P$ as, intuitively, Q should be included in S_P (taking a bigger S_P allows a smaller P_L, which may cause less problems regarding the extension of the LHS). As in previous cases, the algorithm can be iterated.

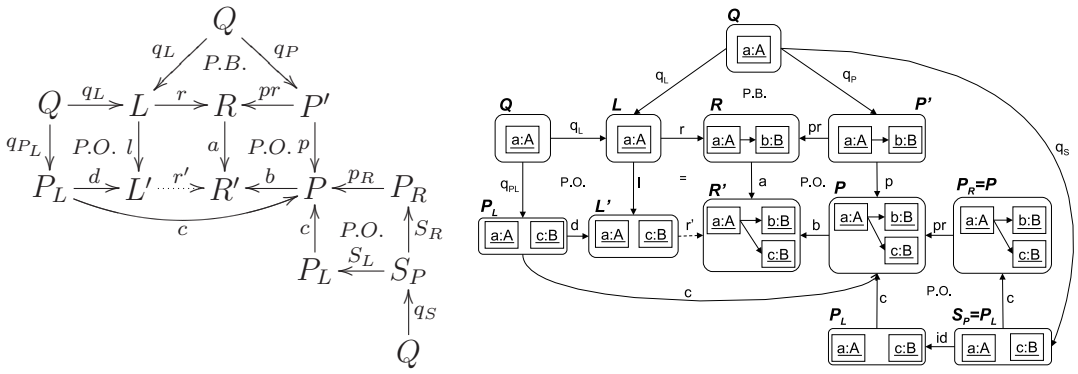

Fig. 19. Extending both Context and Effect of a Rule (left). Example (right).

Example. The right of Fig. 19 shows an example where both the rule's context and its effects are enlarged. In this case, no maximal context was sought, as P_L omits the edge between `a` and `c`. In this particular case, S_P is isomorphic to P_L and P_R to P. Altogether, the resulting rule has an additional node `c` in its context, and then increases its effects by adding an edge between `a` and `c`.

For appropriate categories (like typed graphs or typed triple graphs), we also introduce *abstract patterns*, containing elements of some abstract type from a type graph with inheritance TGI. An abstract pattern ap is equivalent to a set $conc(ap)$ of *concrete patterns*, resulting from all valid substitutions of the abstract types by concrete types in the corresponding inheritance clan.

Sometimes, more than one pattern have to be enforced by one rule. The policy for applying a set of patterns is intentionally left open, and depends on the category instantiating the EG-patterns. Working with non-deleting rules, and taking the restriction that each maximal intersection object should have a non-empty intersection with the RHS of the original rule, makes the application order of patterns irrelevant. Conflict detection mechanisms for other policies are left for future work.

The next section introduces triple patterns as an instantiation of EG-patterns for the category of triple graphs, while Section 8 gives an instantiation for DPO rules (called action patterns).

7 Patterns for the Specification of Static Semantics

This section presents *triple patterns*, an instantiation of EG-patterns for triple graphs, together with a specialization of the previous constructions. We are interested in triple patterns declaring the admissible relations between elements in concrete syntax and semantics models. We present an algorithm that, given an editing rule acting on the concrete syntax only and a set of triple patterns, generates an operational TGG rule that synchronously creates the necessary elements in the target and correspondence graphs. The algorithm is based on the constructions given in the previous section (instantiated for attributed typed triple graphs [17]), thus generalizing the one given in [10]. Symmetrically, the input rule could act on the target graph, and the generated TGG rule would complete the source graph. This could be useful to extend the execution rules acting on the semantic model only (like the ones we generate in next section) to synchronous TGG rules modifying in addition the concrete syntax. As in [32], we can also generate other TGG operational rules: batch rules (i.e. assume that the source elements exist, and then create the target graph elements), rules for creating the correspondence graph given a source and a target graphs, and for checking the validity of the correspondence graph.

Example. We first start by giving an intuition of the algorithm through an example. We use triple patterns in order to specify in a visual, high-level, acausal notation the kind of configurations we want to find in our semantic (syntax) models when certain syntactic (semantic) configurations are met. In this example, triple patterns are triple graphs conforming to the meta-model of Fig. 5. The triple pattern in Fig. 20 depicts the needed structure in the syntactic model for a holder to have a token in the semantic model. In this case, a `Place` in the syntactic model has an associated `PlaceSem` role (a subclass of `Holder`) in the semantic model. Similarly, a `Token` in the syntactic model has a `TokSem` role in the semantic model (a subclass of class `Token` in the semantic meta-model). In the semantic model, a token `decorates` a holder, while at the syntactic level the place `contains` the token.

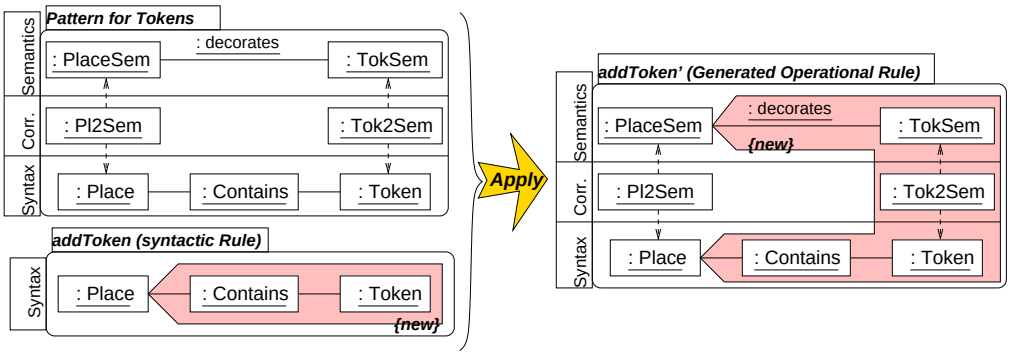

Fig. 20. Applying a Pattern to a Rule.

Fig. 20 also shows a syntactic editing rule ("addToken") creating a token inside a place in the syntactic model. The objective of the algorithm is to obtain the triple rule shown on the right of the figure, incorporating information about the actions to be done at the semantic level, together with the mapping between syntactic and semantic models. We use the algorithm for applying SP-patterns in the previous section, extending both the rule context and its effects.

A triple pattern $P : P_{src} \xleftarrow{ps} P_{corr} \xrightarrow{pt} P_{tar}$ is a triple graph conformant to (typed by) a meta-model triple. Formally, given a triple pattern P and a triple graph G, we say that G satisfies P (written $G \models P$) if an injective triple graph morphism $m : P \to G$ exists. The following algorithm describes the application of a set of patterns to a non-deleting standard rule, resulting in one triple rule. The algorithm can be easily modified for its application to deleting (and non-creating) rules by reversing the roles of L and R.

`Apply(STP: SetOfTriplePattern, rl: Rule): TripleRule`

Let $STP = \{P^i\}_{i \in I}$ be a set of triple patterns $P^i : P^i_{src} \xleftarrow{ps^i} P^i_{corr} \xrightarrow{pt^i} P^i_{tar}$ and rl a non-deleting standard rule $rl : L \xleftarrow{l} K \xrightarrow{r} R$ with $L = K$, and which can therefore be written as $rl : L \xrightarrow{r} R$. The application of STP to rule rl results in a triple rule rl' as follows:

(1) Initialize the triple rule rl' by copying rl in the source part of rl'. The resulting triple rule is written as $rl' : L_{rl'} \xrightarrow{r'} R_{rl'}$, where r' is a triple graph morphism (see Fig. 21).

$$rl' = \begin{tikzcd}
L_{tar} = \emptyset \arrow[r, "r'_{tar}=\emptyset"] & R_{tar} = \emptyset \\
L_{corr} = \emptyset \arrow[u, "lt=\emptyset"] \arrow[r, "r'_{corr}=\emptyset"] \arrow[d, "ls=\emptyset"] & R_{corr} = \emptyset \arrow[u, "rt=\emptyset"'] \arrow[d, "rs=\emptyset"] \\
L_{src} = L \arrow[r, "r'_{src}=r"] & R_{src} = R
\end{tikzcd}$$

Fig. 21. Initialization of Triple Rule rl'.

(2) $\forall$ SP-pattern $P^i : P^i_{src} \xleftarrow{ps^i} P^i_{corr} \xrightarrow{pt^i} P^i_{tar} \in P$, apply P^i to the triple rule rl', extending the rule effects and also the context, according to the procedure given in Section 6, see also Fig. 19. The maximal intersection object P' should be chosen as the biggest triple graph $P' \hookrightarrow R$ such that $P'_{src} = p^i_{src}$, that is, the whole source part of the pattern should be found in R. The triple graph P_L should be chosen as the restriction $P^i_{tar}|_c \xleftarrow{ps|_c} P^i_{corr}|_c \xrightarrow{pt|_c} P^i_{src}|_{L_{src}}$ of triple pattern P^i to elements in L_{src}, where $P^i_{tar}|_c$ contains those elements in P^i_{tar} related to elements in L_{src} or not related to any element in the source graph.

Fig. 22 shows an example of the execution of the procedure to the rule and

pattern shown to the left of Fig. 20. Thus, R' is a triple graph that connects the source elements with the target ones according to P. Moreover, L' is built by connecting the elements in the source part of L according to P_L.

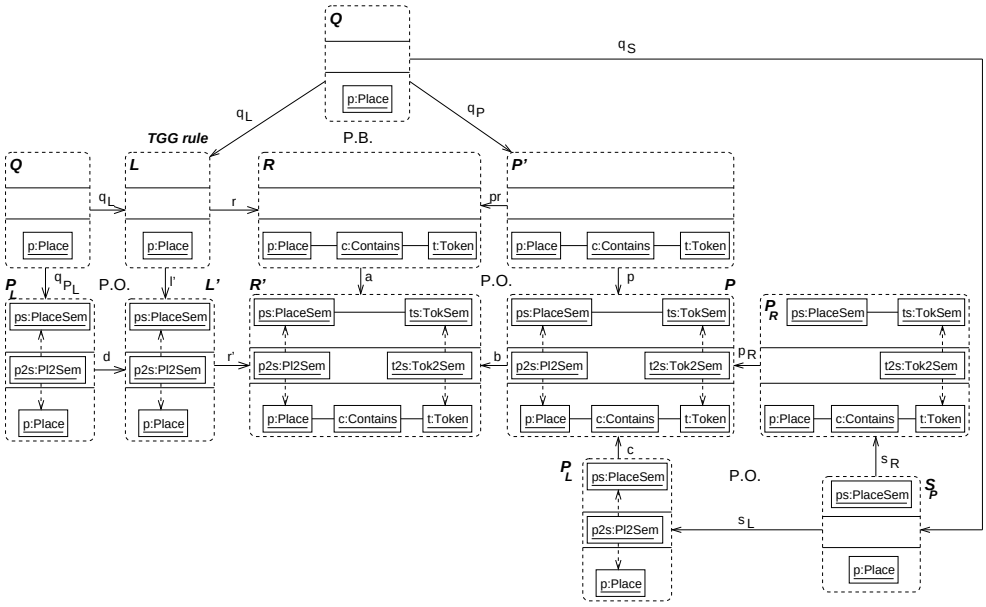

Fig. 22. Applying the Pattern for Tokens.

Fig. 23 shows additional patterns for the Petri nets example. According to the left pattern, output places of a transition in the syntactic graph are post-condition `PlaceSem` objects for the `TransSem` object associated with the transition. The pattern to the right models the correspondence for input places. By applying these patterns to the editing rule to the left of Fig. 24 (twice the pattern for post-conditions, and once that for tokens), we obtain the operational TGG rule shown to its right.

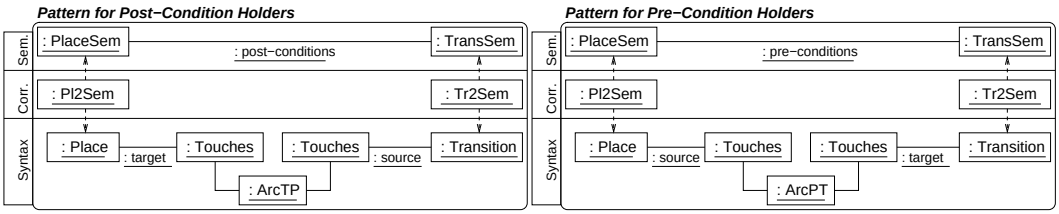

Fig. 23. Additional Patterns for the Example.

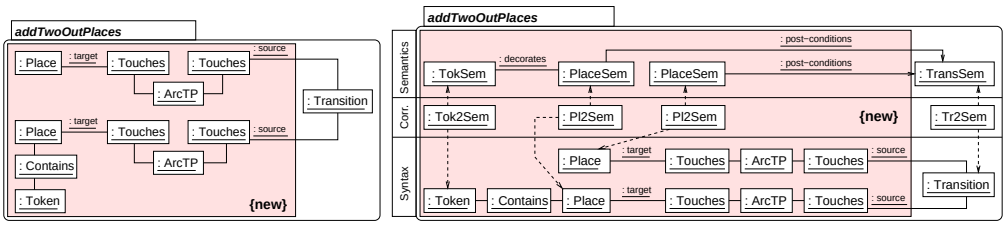

Fig. 24. Syntactic Rule (left). Derived TGG Rule (right).

The advantage of these patterns is that they are specified once, and can then be applied to complex syntactic rules. The DSVL designer does not have to modify each syntactic rule by hand in order to add the semantic information, but only to specify the patterns once. Moreover, the patterns do not have to take into account which elements are created and which are already existing, as this is specified in the standard rules to which they are applied. Thus, the pattern may be used in several ways (i.e. in parts of the rule which are newly created or in existing ones). In our example, the same patterns are used to build the semantic model from the concrete syntax (by applying them to editing rules), and to extend the execution rules acting on the semantic model only (see next section) to TGG rules synchronously modifying both the semantic model and the concrete syntax.

7.1 Abstract Triple Patterns

Triple patterns containing "abstract objects" (i.e. instances of abstract types) may be applied to rules also containing "abstract objects" (i.e. abstract rules). When looking for a match from an abstract pattern to a rule, abstract objects in the pattern can be matched with objects of more concrete types in the rules. We shortly indicate by $conc(r)$ the set of concrete rules obtained by replacing abstract objects in r with all possible objects in the corresponding clans [9]. Following the ideas in [10], the previous algorithm is thus modified:

`AbstractApply(STP: SetOfTriplePattern, r: Rule): SetOfTripleRules`

Let $STP = \{P^i\}_{i \in I}$ be a set of (abstract) triple patterns, with $P^i : P^i_{src} \xleftarrow{ps^i} P^i_{corr} \xrightarrow{pt^i} P^i_{tar}$ and $r : L \xrightarrow{r} R$ a non-deleting rule. The application of STP to r results in a set of triple rules $RG' = \{r'_j\}$, as follows:

(1) Set $RG' = \emptyset$.
(2) Let $RG = conc(r) \cup \{r\}$ be the set of concrete rules equivalent to r and r itself.
(3) Let STP^c be the set of concrete patterns equivalent to the patterns in STP.
(4) $\forall r_k \in RG$, $RG' = RG' \cup Apply(STP^c, r_k)$. That is, we apply the concrete patterns to each concrete rule.
(5) $\forall r' \in RG'$: if $\exists r'' \in RG'$ s.t. r' is more concrete than r'' ($r' \preceq r''$) then $RG' = RG' \setminus \{r'\}$. That is, we eliminate rules "subsumed" by others (same structure, equal or more concrete types).
(6) $\forall r' \in RG'$: if $\exists r'' \in RG'$ s.t. $r''_{src} \preceq r'_{src}$ then add a NAC to r' with all the nodes in r'' that are refinements of nodes of r', where r'_{src} is the standard rule resulting by taking the source graphs of triple rule r'.

Thus, the idea is to first generate the set of concrete rules equivalent to the

abstract rules (step 2), as well as the set of concrete patterns equivalent to the abstract ones (step 3). After applying the patterns, we eliminate redundant rules (step 5) and finally, NACs are added to the generated rules (for the source graph), so that they cannot be applied to more concrete types if a refined rule was also generated. For efficiency reasons, an implementation of this algorithm would not generate all concretizations of rules and patterns at steps (2) and (3), but work at an abstract level, for example using clan morphisms [9], which take into account the inheritance relations in the meta-models. For further details on the algorithm, the reader is referred to [10].

8 Patterns for the Specification of Execution Semantics

This section describes an instantiation of EG-patterns for the category of DPO rules (called *action patterns*) and how the constructions in Section 6 have to be specialized. We use action patterns for the construction of execution rules by means of meta-rules. Each meta-rule, associated with an editing rule, is used to incrementally construct an execution rule describing the semantics of a particular *active* element of the model (e.g. a transition). However, to avoid writing each meta-rule by hand, we propose to exploit a set of *action patterns* to describe semantics. We present a procedure to generate a meta-rule, starting from a set of patterns and an editing rule.

A *type system for patterns* over a type graph with inheritance $TGI = (TG, I)$ is a construct $TSP = (TGI,\ tr,\ \sigma)$, where $tr \in N_T$ is a designated node type whose semantics is described by the action patterns (e.g. a transition in the case of a Petri net). In addition, $\sigma \subset TGI$ is a subgraph of types (with inheritance) relative to the execution mechanism, with $tr \in \sigma_{N_T}$ (the set of nodes in σ). Elements in σ are needed in order to express the operational semantics of the language (e.g. places and arcs). An *action pattern* over TSP is a DPO rule $ap : L^a \xleftarrow{l^a} K^a \xrightarrow{r^a} R^a$ such that $\bigcup_{n \in X^a_N} type(n) \cap \sigma_{N_T} \neq \emptyset$, for $X = \{L, K, R\}$, where X_N is the set of nodes of graph X. Thus, an action pattern is a DPO rule where some elements have types in σ. As in Section 7, we admit *abstract action patterns* containing elements with abstract typing.

Fig. 25 shows two abstract action patterns for a general transition semantics. Pattern *get* deletes a token from a pre-condition holder, i.e. it removes both the token and its association with the holder. In a similar way, pattern *put* adds a token and an association with a post-condition holder.

The *get* and *put* patterns are abstract and therefore highly reusable, as they are applicable to any language with transition-based semantics, for example to place/transition Petri nets (see the meta-model triple to the left of Fig. 5). The type system for the patterns is based on the semantic meta-model (the

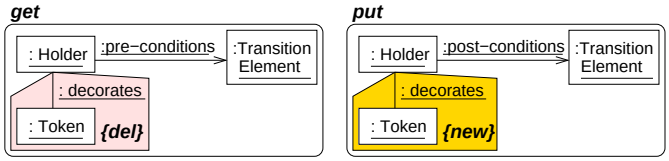

Fig. 25. Action Patterns for Transition Based Semantics.

upper one) in Fig. 4, and the subclasses added by each particular language. In all cases, the designated type *tr* is `TransitionElement`. For the example of Petri nets, σ contains the classes in the semantic model of Fig. 5, except class `Token` and the `decorates` association. For some classes of Petri nets, where tokens with identities are used, one can introduce a *move* pattern that does not remove or insert tokens, but only transfers the *decorates* association connecting the token from one holder to another.

8.1 Generating the Meta-Rules

We present an algorithm that, given a set of action patterns and an editing rule, generates a meta-rule that updates the execution rule associated with a transition element. The algorithm is based on the constructions for EG-patterns in Section 6. To provide intuition, we illustrate how the action patterns in Fig. 25 are applied to the editing rule on the left of Fig. 26 to obtain the meta-rule on its right.

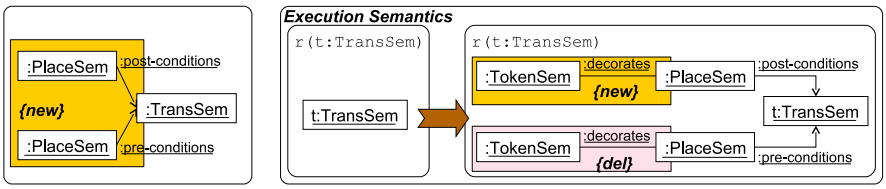

Fig. 26. Editing Rule (left). Associated Meta-Rule (right).

As the editing rule adds a pre- and post-holder to an existing transition, the associated meta-rule must update the execution rule by adding the semantics of an additional pre-holder and post-holder. Hence, the meta-rule should identify the transition in the execution rule and modify it by enlarging the LHS with the pre- and post-holders, together with a token in the pre-holder. Then, the RHS is enlarged with the pre- and post-holders, the deletion of the token in the pre-holder and the addition of the token in the post-holder.

In a situation as depicted on the left of Fig. 27, the editing rule (and therefore the associated meta-rule) has been fired twice for the transition. This produces the execution rule shown to its right. As previously mentioned, the rule application mechanism initializes the match of the execution rule (using its parameter) with the transition of the editing rule. Thus, such execution rule can only be applied at the transition in the semantic model to its left.

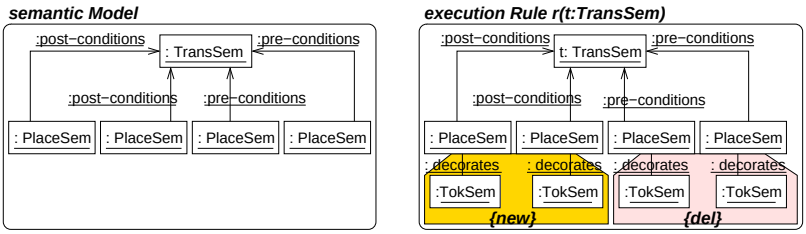

Fig. 27. Semantic Model and Resulting Rule.

Let $TGI = ((N_T, E_T, s^T, t^T), I)$ be a type graph with inheritance, $TSP = (TGI, tr, \sigma)$ a type system for patterns over TGI, $AP = \{ap_i : L_i^a \xleftarrow{l_i^a} K_i^a \xrightarrow{r_i^a} R_i^a\}_{i \in I}$ a set of action patterns over TSP and $e : L \xleftarrow{l} K \xrightarrow{r} R$ an editing rule typed over TGI. The *application* of AP to e produces a meta-rule for each transition element in e, according to the following algorithm.

`Apply(AP:SetofActionPattern, e:Editing Rule, tsp:TypeSystemPattern): SetofMeta-rule`

(1) Initialize the set of meta-rules, $MRS = \emptyset$.
(2) Set $AP^c = \bigcup_{ap \in AP} conc(ap)$, i.e. all concretizations of the action patterns.
(3) $\forall t \in R|_{clan(tr)|_{\sigma_{N_T}}}$ ($R|_{clan(tr)|_{\sigma_{N_T}}}$ is the RHS of e restricted to subtypes in σ_{N_T} of the designated node type tr):
- Initialize the meta-rule mr_t as follows: $L^s = K^s = R^s = K|_t$, where $K|_t$ is the kernel of the editing rule restricted to node t. $L'^s = K'^s = R'^s = sub(R,t)|_\sigma$, where $sub(R,t)|_\sigma$ is the smallest connected subgraph of R containing node t and no other element t' with type in $clan(tr)$, restricted to types in σ. The meta-rule thus becomes: $mr_t = (L^s \xleftarrow{id} K^s \xrightarrow{id} R^s) \rightarrow (L'^s \xleftarrow{id} K'^s \xrightarrow{id} R'^s)$.
- $\forall ap_j : L^a \xleftarrow{l^a} K^a \xrightarrow{r^a} R^a \in AP^c$, apply ap_j to meta-rule mr_t extending the meta-rule effects, according to the first procedure given in Section 6.
- The parameter of the meta-rule's LHS and RHS are $K^s|_t$ and $K'^s|_t$.
- Add mr_t to the set MRS.

(4) return MRS.

As in the algorithm of Section 7.1, an efficient implementation would not generate the concretizations of the patterns, but work at the abstract level. Fig. 28 shows the execution of the algorithm for patterns *get* and *put* and the editing rule in Fig. 26. This rule contains a single transition, hence one meta-rule is generated. The LHS of the meta-rule is initialized with the unique transition element, the RHS with the smallest connected subgraph of the editing rule's RHS that contains the transition element and no other one. In addition, the initialization is restricted to elements with types in σ, which usually does not contain dynamic elements (i.e. tokens). This is necessary as the fact that an editing rule adds or deletes tokens is irrelevant for updating the execution rule. The concretized *get* pattern is applied once. This pattern is like the one

in Fig. 25, but with elements of type `PlaceSem`, `TransSem` and `TokSem` instead of `Holder`, `TransitionElement` and `Token`. After applying *get*, pattern *put* is applied, thus yielding the final RHS. The left and right hand side rules of the resulting meta-rule have the `TransSem` node as parameter (not shown in the figure for simplicity). Hence, the generated meta-rule takes an execution rule with a `TransSem` in its LHS, RHS and parameter, and adds a pre- and a post-condition, together with the appropriate handling of tokens.

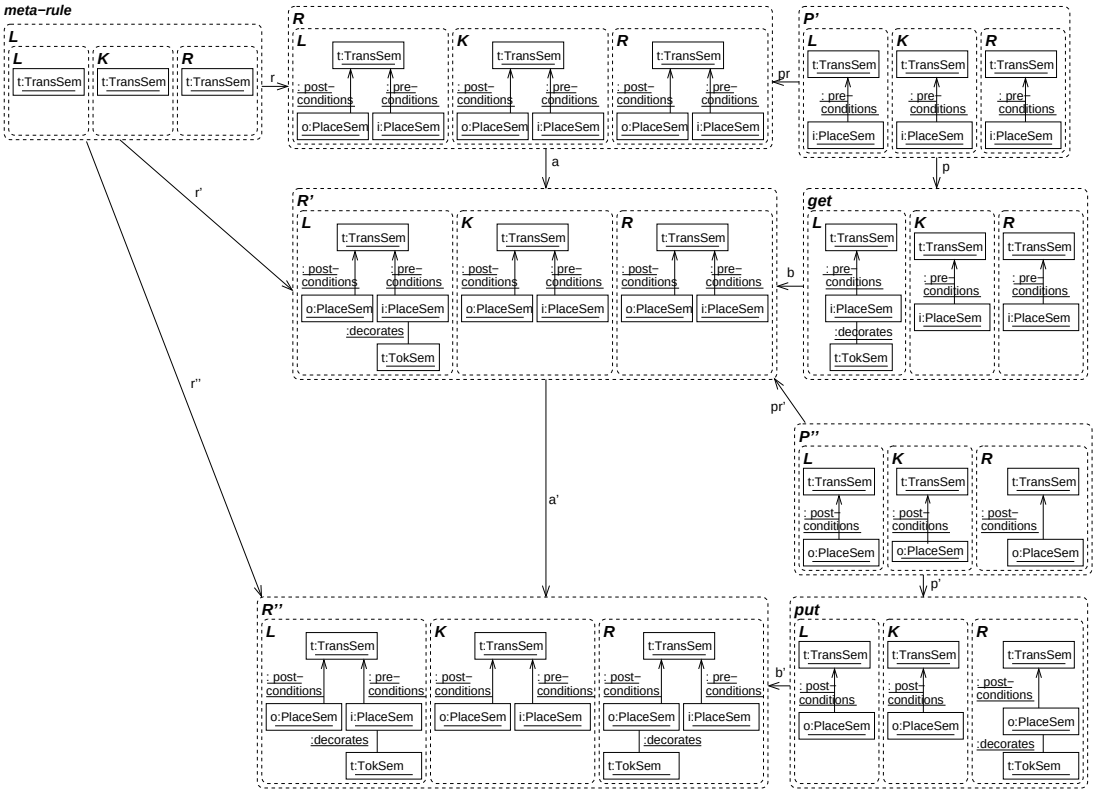

Fig. 28. Applying Patterns *get* and *put*.

Once the execution rules are generated, we can apply the triple patterns presented in previous section to extend them to synchronous TGG rules modifying both the static semantics and the concrete syntax.

9 Examples

We provide examples highlighting the use of patterns in the incremental construction of the abstract syntax and execution semantics of visual languages.

As Petri nets have been used throughout the paper, we just briefly present some further examples of languages with token-holder semantics, to show both the applicability of such semantics and our pattern concept.

State Automata. The left of Fig. 29 shows the meta-model triple for state automata. At the syntactic level, states are both entities – as they can be connected to transitions – and containers, as the current state contains a decoration inside (class `Current`). At the semantic level, states are holders (i.e. they can receive a token, becoming the current state), while transitions are transition elements. For simplicity, we do not consider events in transitions.

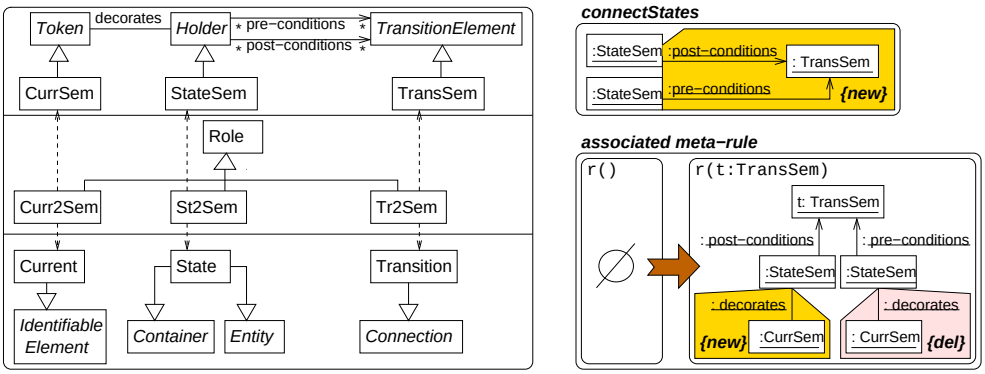

Fig. 29. Meta-Model for State Automata (left). Editing Rule (top right). Generated Meta-Rule (bottom right).

The action patterns *get* and *put* shown in Fig. 25 are valid for automata, as a transition element has exactly one pre- and one post-condition. Hence, *get* removes the token from the current state (a pre-condition) and *put* inserts it into the post-condition holder. The type system is given by the semantic meta-model in Fig. 29, except class `Token` and its child, and association `decorates`. The right of Fig. 29 shows an editing rule and the generated meta-rule. As the transition element is created when connecting the two holders (i.e. states), the meta-rule creates the transition element in its RHS.

In this example, the DSVL designer writes the concrete syntax of the language, puts it in correspondence with the semantic model (i.e. builds the meta-model triple in Fig. 29, together with the triple patterns) and builds the editing rules for the environment. By applying the action patterns, the execution rules acting on the semantic model are generated. Then, one applies triple patterns to editing rules (to obtain TGG rules updating the semantic model) as well as to the execution rules (to obtain TGG rules to update the concrete syntax model). Otherwise, these two sets of rules would have to be created by hand. Given a semantic variety, it should also be possible to automatically derive a set of triple patterns from a meta-model triple. We leave this for future work.

WorkFlow. Fig. 30 shows an excerpt of the meta-model for a simple workflow language, in the style of [34]. Two kinds of blocks – parallel and sequential – exist, playing the roles of both transition elements and holders. Parallel blocks are amenable to incremental semantics, as they require a token in each incoming block for firing, and add a token in each outgoing block.

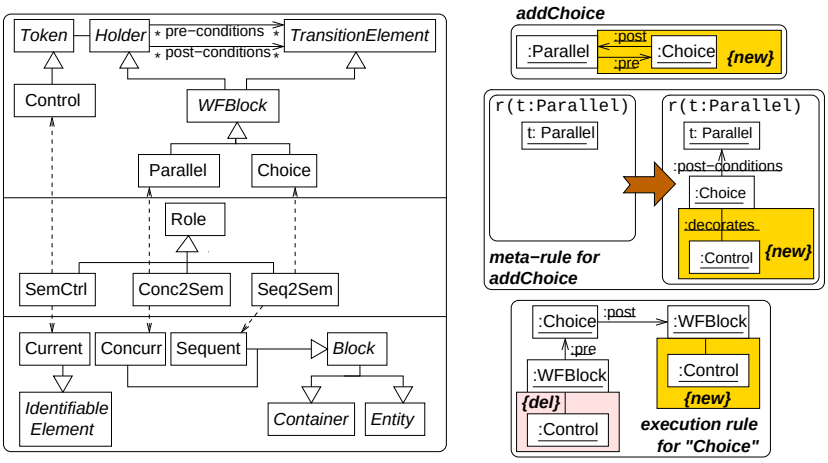

Fig. 30. Meta-Model for Workflow Language (left). Editing Rule (top right). Generated Meta-rule (middle right). Execution Rule for `Choice` (bottom right).

An example editing rule is shown in the upper-right corner of Fig. 30. This rule adds a choice as a post-condition for the parallel block, while the latter becomes a pre-condition for the choice. Below the editing rule the generated meta-rule for *addChoice* is shown. Choice blocks have a sequential semantics: they take one token from one of the incoming blocks (randomly chosen), and put the token in one of the outgoing ones (also randomly chosen). No incremental construction is needed for choice blocks, and the global execution rule in the lower-right corner of Fig. 30 is enough. This rule is abstract (equivalent to four normal rules), as we do not care about the explicit type of the incoming or outgoing blocks. Thus, in this case, we need the type system TSP to include `Choice` in the determination of the context of the execution rules for `Parallel`, but we avoid the generation of a meta-rule for the class `Choice` which does not need incremental semantics. Therefore, TSP includes in TG all types in the upper part of the meta-model triple of Fig. 30, but excludes `Choice`, `Token` and `Control` from σ_{N_T}.

In this example the DSVL designer can reuse the action patterns for the token-holder semantics. However, the DSVL semantics for this case is not entirely covered by the patterns, but has to be refined by adding a non-incremental rule (to avoid the parallel semantics of the choice blocks).

In Discrete-Event Simulation Systems (DESS) [14,31], a number of entities generate events to be processed at a future time. These events are stored (ordered by their execution time) in a so called *future event queue*. The simulator proceeds by taking the first event in the queue and executing it. In the event-scheduling approach, the execution of an event follows a specification and gives rise to the scheduling of further events.

Event Graphs [31] are a classical example of event-scheduling DESS. A simple model is shown in Fig. 31, with four types of events and some causality relations. An event graph model does not have to worry about the event queue, as this is a mechanism for execution, which does not belong to the modelling phase. Other discrete simulation languages, such as Time Transition Petri nets or Process-Interaction notations [14], have similar execution semantics.

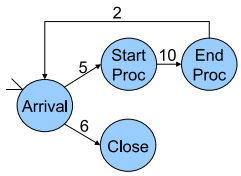

Fig. 31. An Event Graph.

The meta-model triple on the left of Fig. 32 extracts the DESS semantics and relates it to the concrete syntax of the particular DESS formalism. An executable DESS model needs some event specification and an event queue for its execution. The specification defines event types, and is made of a number of event types related through `Schedules` objects, annotated with the time after which an event of the given type will occur. Events in the event queue contain the time at which they have to be executed, and are instances of some event type, as depicted by the association `instanceOf`. Each event in the queue points to the next one. A `Scheduler` entity takes care of ordering new events in the queue. The type system for the action patterns contains classes `Schedules` and `EventSpec` (which is the distinguished active element) together with associations `source` and `target`. The concrete syntax part of the meta-model triple assumes that event specifications can be produced using any `referable element`, but the scheduling of new events is specified via directed connections (i.e. we assume a graph-like language). The right of Fig. 32 shows the specialization for Event Graphs.

The syntactic rule on the left of Fig. 33 creates new event specifications (i.e. event types). The triple pattern in the middle shows the relation between event specifications in the concrete syntax and the semantic model. The action pattern on the right describes part of the execution semantics of the newly created event type. Thus, if an event of this type is the first one in the queue, the execution rule deletes it and moves the queue pointer to the following one.

Once the event specification has been created, a possible syntactic action is

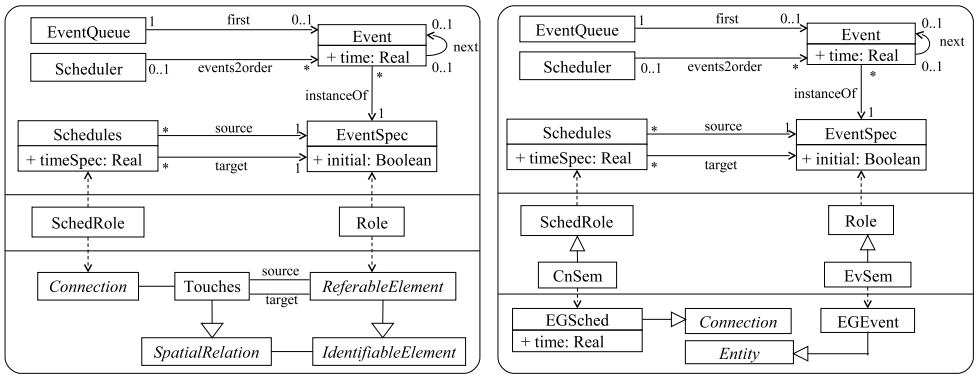

Fig. 32. Meta-Model Triple for DESS (left). Specialization for Event Graphs (right).

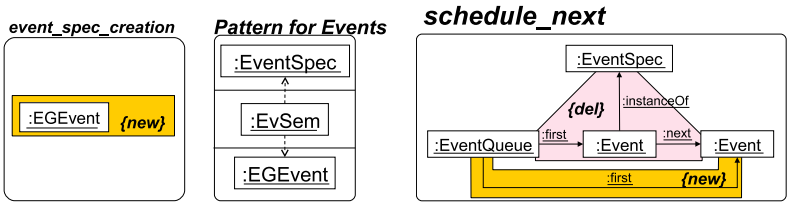

Fig. 33. Editing Rule for Creating Event Types (left). Triple Pattern for Event Specifications (middle). Action Pattern (right).

the scheduling of other event specifications when an event of the given type is executed. This is modelled by the syntactic rule shown on the left of Fig. 34. The (attributed) triple pattern in the middle describes the relation between connected events in the concrete syntax and in the semantics. The action pattern on the right describes the execution semantics for event scheduling. Thus, the modification to the meta-rule produced by this pattern allows changing the execution rule associated with the source event specification, so that it creates a new event of the type specified by the schedule object's target. The new event is created and linked to the scheduler object, the behaviour of which is handled by the non-incremental rules shown in Fig. 35.

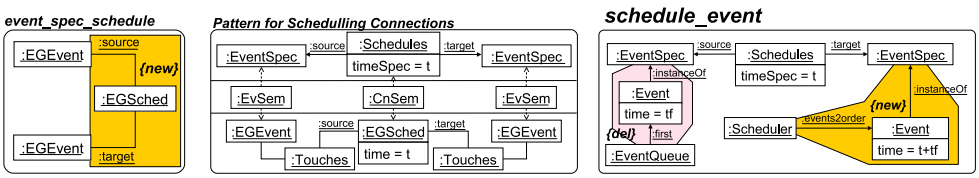

Fig. 34. Editing Rule for Scheduling new Events (left). Triple Pattern (middle) and Action Pattern (right).

The set of non-incremental rules shown in Fig. 35 initialize the simulation (*init*) and model the behaviour of the scheduler (rules *insert* and *insert_first*). Rule *init* creates the scheduler and the event queue, initialized with the initial

and final events. The former is an instance of the initial `EventSpec`, the latter is used to ease the management of the queue.

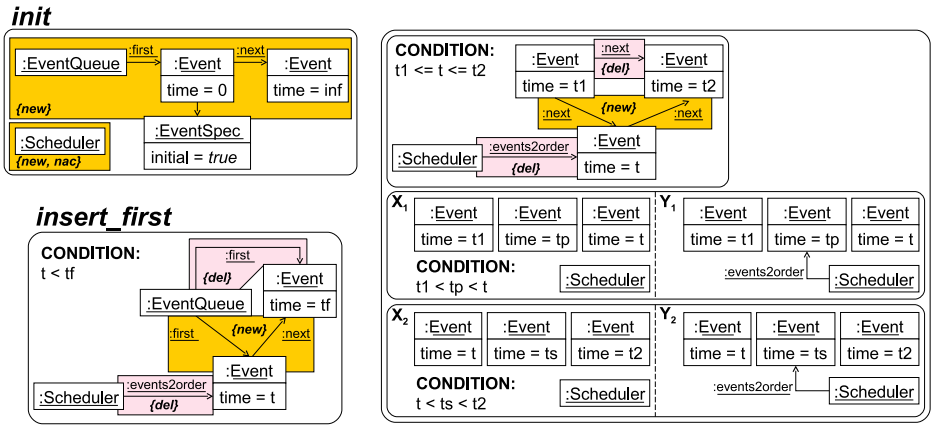

Fig. 35. Non-Incremental Rules.

The other two rules in the figure insert new events in the queue. These events are linked to the scheduler by means of an `events2order` relation. Rule *insert_first* inserts a new event in the queue if this event is the first one (i.e. it has lower time than the first scheduled one). Rule *insert* schedules an event in the queue, ordered by time. The LHS selects two consecutive events in the queue so that the event is added in between, and two application conditions check that the time ordering is preserved. The first condition ensures that, if there is an event with a time tp between the time of first event selected ($t1$) and the time of the event to be scheduled (t), then such event is not in the queue, but is being also scheduled. The second condition performs a similar check with the second selected event by the LHS.

Note that the non-incremental rules of Fig. 35 are general for any DESS language (i.e. not specific to Event Graphs), and therefore, together with the DESS patterns, will be reusable for any DSVL with DESS semantics. As an example, Fig. 36 shows the execution rule automatically generated for event "Arrival" in the model of Fig. 31.

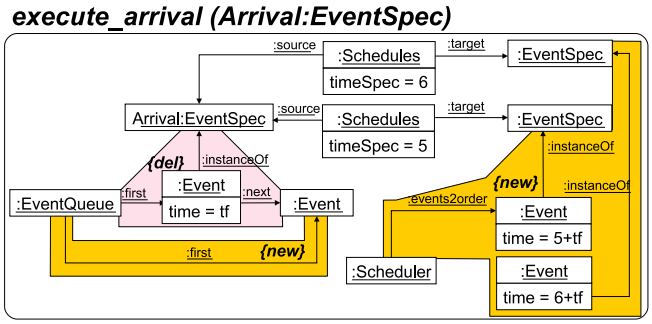

Fig. 36. Generated Execution Rule for Event Type "Arrival".

Thus, with our approach, creating a DSVL with DESS semantics implies:

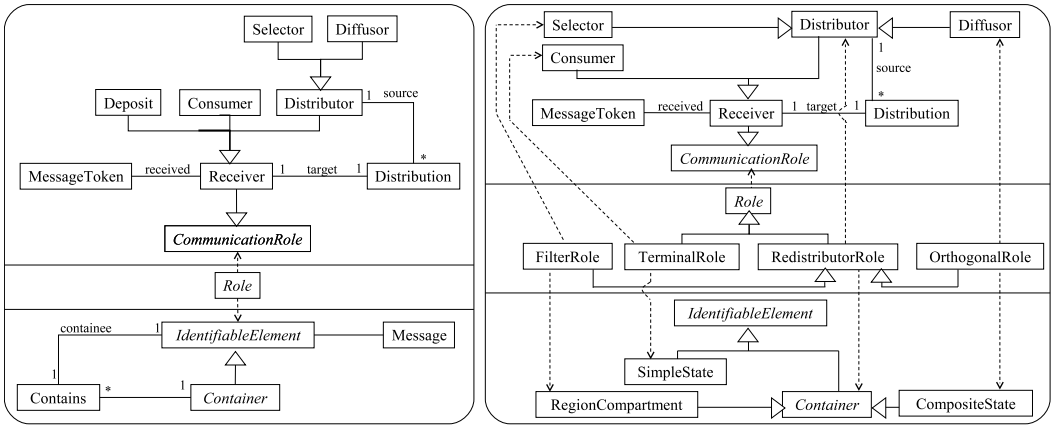

Fig. 37. Relating Meta-Models for Communication and Containment-Based Visual Languages (left). Specialization to UML State Machines (right).

designing the concrete syntax and putting it in correspondence with the semantic model (bulding the meta-model triple and the triple patterns), as well as building the editing rules. In this case, the execution rules are generated from the action patterns, but the DESS semantics also include the set of rules in Fig. 35, which are also reused for the particular DSVL. As in the examples of previous subsection, triple patterns are used to extend the editing rules and the execution rules into TGG rules.

9.3 Visual Languages for Communication Structures

Visual languages for specifying communication processes typically allow an explicit representation of a communication infrastructure, plus the definition of a communication protocol. The elements in the infrastructure are responsible for receiving and sending messages according to the protocol. In general, messages are exchanged only among elements enjoying some matching visual property, such as 1) residing at different ends of a same connector; 2) occupying adjacent cells in a grid; 3) sharing a same container; 4) being within distance from a source element. Communication elements can have different roles in different protocols. The meta-model triple on the left of Fig. 37 describes the relations between communication roles and their visual counterparts depicting a communication infrastructure in terms of the containment relation.

`Identifiable Elements` play the role of `Receiver` of a `MessageToken`, represented as a property of the receiving element. `Containers` act as `Distributors` of the messages they receive towards their *containees*. Elements which are not containers are message `Consumers`. The existence of a `Contains` relation in the diagram indicates the existence of a `Distribution` relation at the semantic level. Moreover, distributors can act as `Selectors`, transferring the message

exactly to one containee, or `Diffusors`, transferring a copy of it to all of them. Finally, a `Deposit` role is played by a container which maintains messages, for receivers to retrieve them according to some other visual relation.

As an example, the right of Fig. 37 shows a simplified version of the model for the graphical representation of UML State Machines. We only consider the containment-based aspect of a machine and its semantics in terms of distribution of event-messages to internal states, seen as processes started by the reception of a message (or event) at the containing `State` interface. A `SimpleState` is a `State` for which the `isSimple` attribute has value *true*. These can only be `Consumers` of a message, as indicated by the morphisms from the `TerminalRole` class. They have a distinct representation from that of `Composite States`, which in turn are distinguished in `Orthogonal` or not. Composite states contain `RegionCompartments`, which may in turn contain other composite or simple states. An orthogonal composite state contains at least two regions, separated by a dashed line, while a non-orthogonal state contains only one region. A region always acts as a `Selector`, delivering the message only to the currently active one inside it. A composite state always acts as a `Diffusor` of the message to all its internal region compartments. This is represented by the different specializations of the `CommunicationRole`. We omit initial and terminal states, as these do not play a role in the communication semantics, and do not consider any special representation for a message.

Based on the meta-model triple, one can incrementally construct a communication semantics, while designing a state machine through syntactic graph transformation rules. With each `Composite State` creation, a corresponding receiver structure is added to the semantics. Elements can be inserted only inside containers (the presence of an outermost container being an axiom); this generates a new instance of the `Distribution` relation and creates the correspondence with the inserted element. We use here only the most specific role that can be assigned to an element. Hence, when inserting a composite state, one must specify whether it is orthogonal or not. An orthogonal state is created with two empty regions, and new regions can then be added.

Fig. 38 describes a set of triple rules and a concrete syntax presentation: regions are added to the bottom of a composite state and the size of the container is expanded to accommodate the new region, as prescribed by rule $r1$. Rule $r2$ shows how to add simple states to a region compartment. Triple rules are obtained from syntactic ones by specializing the abstract triple pattern on the left of Fig. 39, as well as the abstract action pattern on the right, to the most concrete patterns for `Selectors`, `Diffusors` and `Consumers` roles.

Finally, a communication semantics can be defined exploiting a pattern analogous to that for token-based execution, where messages are used as tokens to be associated with their possible receivers. To this end, the abstract ac-

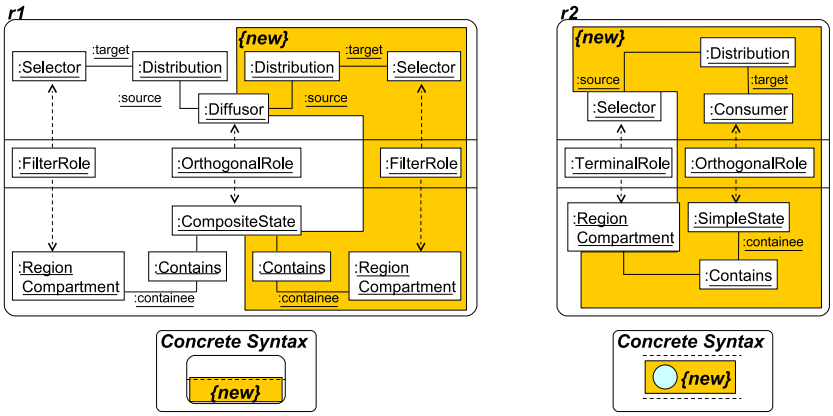

Fig. 38. Rules for Syntax Directed Editing and Incremental Semantics Definition for Communication in State Machines.

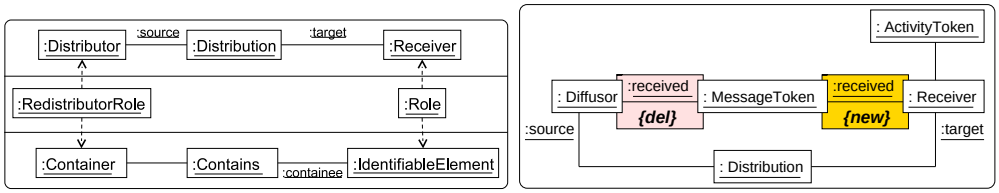

Fig. 39. Triple Pattern for Connecting Containment and Communication Semantics (left). Basic Action Pattern for the Communication Process (right).

tion pattern on the right of Fig. 39 prescribes that a message received by a `Diffusor` must be delivered to any active `Receiver` with which it maintains a `Distribution` relation. The `ActivityToken` element in the abstract pattern represents the condition of being active of a state according to the execution semantics of UML state machines and is a specialization of `Token` from the meta-model for the transition variety, complementing the communication one. In particular, the presence of active tokens in the receivers is constrained by the concurrent semantics of UML state machines.

10 Extensions of Enforced Generative Patterns

In this section we provide additional extensions of EG-patterns: simple negative patterns, composite patterns and patterns with application conditions.

10.1 Simple Negative Patterns

A *simple negative EG-pattern* (short *SN-pattern*) is an object N in a (w)aHLR category. An object H satisfies N, written $H \models N$, if $\nexists m\colon N \to H$ injective

morphism. Thus SN-patterns forbid the application of a rule if the resulting object after the direct derivation does not satisfy the pattern. Therefore, SN-patterns are like negative post-conditions in the graph transformation approach [12]. They can be transformed into pre-conditions, using the well-known procedure to generate post-conditions from constraints and advance post-conditions into pre-conditions [12,19].

The scheme of the construction is shown on the left of Fig. 40. We select a maximal intersection object N_i' and compute the pushout object S_i (square (2)). The object S_i represents a possible result for rule application where pattern N is found (and therefore not satisfied by S_i). Thus, in order to forbid a direct derivation yielding S_i, we construct a negative application condition N_i by applying rule p backwards. This is done by calculating the pushout complement object N_i (i.e. (1) is a pushout). Thus, we obtain a NAC $n_i \colon L \to N_i$. If the pushout complement does not exist, then no NAC has to be added, since S_i cannot be produced. The construction is repeated for each maximal intersection object N_i'. This construction is similar to the one in [12,19]; however, we take a maximal intersection object, instead of just all possible ones. This avoids producing NACs that are subsumed by others.

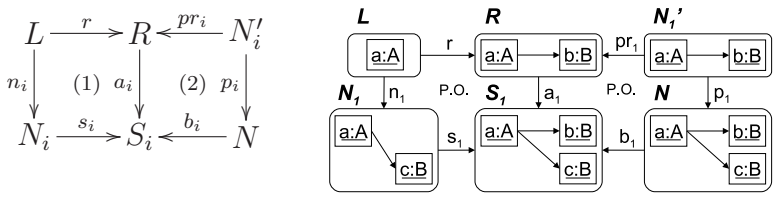

Fig. 40. Translating a *SN-pattern* N into NACs (left). Example (right).

Example. The right of Fig. 40 shows the application of an SN-pattern to a rule. There is no need for further iterations, as there is only one maximal intersection object. Taking a smaller intersection object would have resulted in a weaker NAC, subsumed by N_1. For example, by taking N_1' with just one node of type A, we would obtain a S_1 with three Bs and a NAC with two Bs, which is a weaker NAC than the one we have obtained.

There are two differences with the constructions given in [12]: first, as we only deal with injective morphisms, the calculation of the post-conditions is simpler. Second, we use maximal intersection objects N_i' and not any gluing of R and N. This produces fewer, stronger pre-conditions. As we are interested in constraining the objects produced by R, in our case N_i' cannot be empty.

Composite positive patterns (*CP-patterns*) have the form $X \xrightarrow{x} Y$ (with x injective), meaning: "if X is found, then Y should be found as well". In a similar way, composite negative patterns (*CN-patterns*) of the form $X \xrightarrow{n} N$ mean "if X is found, then N must not be found".

An object G satisfies a CP-pattern $pp : (X \xrightarrow{x} Y)$, written $G \models pp$ iff the diagram to the left of Fig. 41 commutes, or $\nexists X \xrightarrow{m} G$. In this latter case we also say that G trivially satisfies pp. Similarly, an object G satisfies a CN-pattern $np : (X \xrightarrow{n} N)$, written $G \models np$ iff in the diagram on the right of Fig. 41 there is no $N \xrightarrow{t} G$ such that $t \circ n = m$, or $\nexists X \xrightarrow{m} G$. In this latter case we also say that G trivially satisfies np.

$$\begin{tikzcd} X \arrow[rr, "x"] \arrow[dr, "m"'] & & Y \arrow[dl, "y"] \\ & G \arrow[u, phantom, "="] & \end{tikzcd} \qquad \begin{tikzcd} X \arrow[rr, "n"] \arrow[dr, "m"'] & & N \arrow[dl, "t"] \\ & G \arrow[u, phantom, "="] & \end{tikzcd}$$

Fig. 41. Satisfaction of *CP-pattern* (left). Satisfaction of *CN-pattern* (right).

The application of composite patterns to rules is similar to that shown in Section 6, but involves one more step, making an extra pushout.

CP-patterns. The left of Fig. 42 shows the application of a CP-pattern for the case in which pattern X, hence its consequence Y, must be enforced. The procedure is similar to that for SP-patterns (see Section 6), but after enforcing X through the pushout object R', we have to enforce Y by calculating the pushout object R''. Note that X_L and X_R play the same role as objects P_L and P_R in Fig. 19. X_R is the part of X that is enforced by simply creating the missing elements, whereas X_L is enforced by extending the rule's left-hand side. As the CP-pattern $pp : (X \xrightarrow{x} Y)$ can also be trivially satisfied if no X is found in the host graph, another rule resulting from the procedure for SN-patterns (taking only X as SN-pattern) has to be generated.

CN-patterns. In order to apply a CN-pattern $np : (X \xrightarrow{n} N)$ to a rule, we enforce the positive part X and then we forbid N. Enforcing the positive part is done in the same way as for SP-patterns (using the constructions of Section 6). After each iteration, we generate a NAC using the procedure for SN-patterns. The right of Fig. 42 shows a diagram where X is enforced by maximally extending the rule's effects (pushout (1)). Then, the construction of Fig. 40 is applied to generate a NAC $n_i \colon L \to N_i$ (pushouts (2) and (3)). Note that we could have used any of the constructions in Section 6 to enforce X. Moreover, enforcing X may require further iterations, after each of which the procedure for SN-patterns has to be applied (possibly resulting in one or

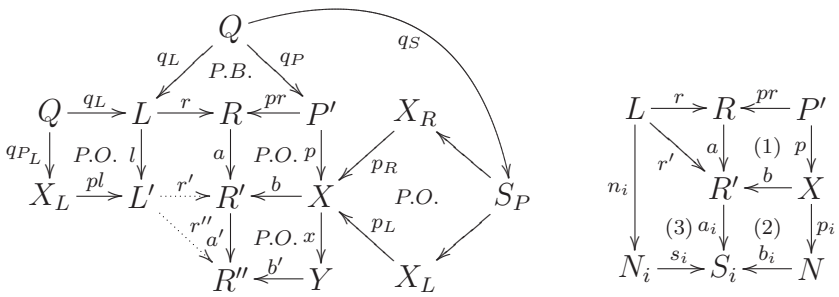

Fig. 42. Extending the Rule Context and Effect According to *CP-pattern* $x\colon X \to Y$ (left). Applying *CN-pattern* $n\colon X \to N$ (right).

zero NACs). Even if no iteration is needed to enforce X (i.e. X is included in R), the SN-pattern procedure must be applied to generate the needed NAC.

10.3 Patterns with Application Conditions

Patterns (of any kind) can also be supplemented with application conditions. These constrain the applicability of the pattern to the rule. We explain the case for SP-patterns and NACs, as the rest is similar. An SP-pattern with NAC is a tuple $(P, n\colon P \to N)$, where $n\colon P \to N$ is a NAC. When the pushout with R' is built, the application is valid only if there is no morphism $x\colon N \to R'$ such that $x \circ n = b$, as Fig. 43 shows.

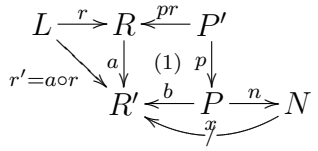

Fig. 43. Application of SP-Pattern with NAC.

11 Conclusions

Patterns are a common way to describe some abstract characteristics of a model, as well as to state some required "shape" for it. The typical way to exploit patterns is therefore through the definition of some morphism from the pattern to an existing model to check whether the latter conforms to the first, i.e. if an instance of the pattern can be found in it. Little support is however provided by this notion for the case in which such a match is not found, for example to identify how conformance can be established.

We have proposed enforced generative patterns as a means to ensure that rules used to produce a model act in such a way that the resulting model conforms to the required shape. In particular, we have shown the relevance of this notion to the development of DSVLs, by showing how the application of suitable sets of patterns to rules of the concrete syntax is sufficient to generate

triple rules, incrementally reconciling the syntax and the static semantics of a visual sentence, and meta-rules, used to incrementally update its execution semantics. Triple patterns can be applied to the execution rules to obtain triple rules updating the concrete syntax. The approach has been demonstrated on a number of known visual languages, showing its potential in the construction of syntax-directed integrated environments for their management.

An important issue for the practical application of EG patterns is management of attributes. We used attributes in triple patterns in Section 9.2 (see e.g. Fig. 34). However, up to now, triple patterns are limited to copy attribute values. The action pattern in Fig. 34 performs a simple attribute computation, and we do not foresee any problem with attribute handling in action patterns. For triple patterns however, we need to restrict to invertible dependencies between attribute values, as the pattern might be applied to the source or the target graphs. Investigating attribute conditions is also left for future work.

We now plan to systematically explore the application of enforced generative patterns to several fields with established notion of patterns. In particular, we see possibilities for the generation of UML-based development environments enriched with support for design patterns in the sense of [15], where basic editing rules could be transformed to generate instances of the patterns. In this way, EG patterns could be used to produce pattern-oriented modelling environments for DSVLs. In this context, activities like pattern discovery and pattern-oriented model redesigns are interesting challenges.

Another interesting development is the introduction of universal quantification into patterns. For example, the basic pattern for creating nodes only if connected to an existing one could be quantified on the latter to ensure that each new node gets connected to all the nodes already in the graph. This could open the way to the definition of a pattern language. This line of research would probably intersect the Local Shape Logic introduced in [27] and the use of patterns as synthetic expression of rules presented in [2].

Acknowledgements. Work supported by the Spanish Ministry of Education and Science, projects MOSAIC (TSI2005-08225-C07-06) and MODUWEB (TIN2006-09678). We thank the referees for their detailed and useful comments, which helped us in improving the paper.

References

[1] T. Baar. Correctly defined concrete syntax for visual models. In *MoDELS/UML 2006*, volume 4199 of *LNCS*, pages 111–125. Springer, 2006.

[2] D. Balasubramanian, A. Narayanan, S. Neema, F. Shi, R. Thibodeau, and

G. Karsai. A subgraph operator for graph transformation languages. In *GT-VMT 2007*, Electronic Communications. EASST, 2007.

[3] L. Baresi and M. Pezzé. Formal interpreters for diagram notations. *ACM TOSEM*, 14(1):42–84, 2005.

[4] P. Bottoni, J. de Lara, and E. Guerra. Action patterns for the incremental specification of the execution semantics of visual languages. In *VL/HCC 2007*, pages 163–170. IEEE CS Press, 2007.

[5] P. Bottoni, D. Frediani, P. Quattrocchi, L.Rende, G.Sarajlic, and D. Ventriglia. A transformation-based metamodel approach to the definition of syntax and semantics of diagrammatic languages. In *Visual Languages for Interactive Computing: Definitions and Formalization*, pages 51–73. IGP Press, 2007.

[6] P. Bottoni and A. Grau. A suite of metamodels as a basis for a classification of visual languages. In P. Bottoni, C. Hundhausen, S. Levialdi, and G. Tortora, editors, *VL/HCC 2004*, pages 83–90. IEEE CS Press, 2004.

[7] P. Bottoni, K. Hoffmann, F. Parisi Presicce, and G. Taentzer. High-level replacement units and their termination properties. *J. Vis. Lang. Comput.*, 16:485–507, 2005.

[8] S. Burmester, H. Giese, J. Niere, M. Tichy, J. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. Tool integration at the meta-model level: the Fujaba approach. *J. Softw. Tools Technol. Transfer*, 6(3):203–218, 2004.

[9] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed graph transformation with node type inheritance. *Theor. Comput. Sci.*, 376(3):139–163, 2007.

[10] J. de Lara, E. Guerra, and P. Bottoni. Triple patterns: Compact specifications for the generation of operational triple graph grammar rules. In *GT-VMT'07*, volume 6 of *Electronic Communications of the EASST*, 2007.

[11] J. de Lara and H. Vangheluwe. Defining visual notations and their manipulation through meta-modelling and graph transformation. *J. Vis. Lang. Comput.*, 15(3-4):309–330, 2004.

[12] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation.* Springer, Berlin, Heidelberg, New York, 2006.

[13] C. Ermel and R. Bardohl. Scenario animation for visual behavior models: A generic approach. *Software and System Modeling*, 3(2):164–177, 2004.

[14] G. S. Fishman. *Discrete Event Simulation. Modeling, Programming and Analysis.* Springer Series in Operations Research, 2001.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[16] H. Göttler. Semantical description by two-level graph-grammars for quasihierarchical graphs. In *WG'79*, Applied Computer Science 13. Carl Hansen Verlag, 1979.

[17] E. Guerra and J. de Lara. Event-driven grammars: Relating abstract and concrete levels of visual languages. *SoSyM*, 6(3):317–347, 2007.

[18] R. Heckel, J. M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In *ICGT 2002*, volume 2505 of *LNCS*, pages 161–176. Springer, 2002.

[19] R. Heckel and A. Wagner. Ensuring consistency of conditional graph rewriting - a constructive approach. *ENTCS*, 2, 1995.

[20] E. Kindler and R. Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical Report tr-ri-07-284, Software Engineering Group, Dep. Comp. Sci., Univ. Paderborn, 2007.

[21] M. Koch, L. V. Mancini, and F. Parisi-Presicce. Graph-based specification of access control policies. *J. Comput. Syst. Sci.*, 71(1):1–33, 2005.

[22] A. Konigs and A. Schürr. Tool integration with triple graph grammars - a survey. *ENTCS*, 148:113–150, 2006.

[23] H.-J. Kreowski. A comparison between Petri-nets and graph grammars. In *Graph-theoretic Concepts in Computer Science*, volume 100 of *LNCS*, pages 306–317. Springer, 1980.

[24] B. Pagel and M. Winter. Towards pattern-based tools. In *EuroPLoP'96*, 1996.

[25] F. Parisi Presicce. Transformation of graph grammars. In *TAGT*, LNCS 1073, pages 428–442. Springer, 1996.

[26] F. Parisi Presicce. On modifying high level replacement systems. In *UNIGRA 2001*, ENTCS, 44(4), pages 16–27, 2001.

[27] A. Rensink. Canonical graph shapes. In D. A. Schmidt, editor, *ESOP 2004*, volume 2986 of *LNCS*, pages 401–415. Springer, 2004.

[28] A. Repenning and T. Sumner. Agentsheets: A medium for creating domain-oriented visual languages. *IEEE Computer*, 28(3):17–25, 1995.

[29] M. G. Rhode, F. Parisi Presicce, and M. Simeoni. Formal software specification with refinements and modules of typed graph transformation systems. *Journal of Computer and System Sciences*, 64:171–218, 2002.

[30] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[31] L. Schruben. Simulation modeling with event graphs. *CACM*, 26(11):957–963, 1983.

[32] A. Schürr. Specification of graph translators with triple graph grammars. In *WG'94*, LNCS, pages 151–163. Springer, 1994.

[33] G. Taentzer. *Parallel and Distributed Graph Transformation. Formal Description and Application to Communication-Based Systems (PhD. Thesis)*. Shaker Verlag, 1996.

[34] W. van der Aalst, A. ter Hoefstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Data Bases*, 14(3):5–51, 2003.