Action Patterns for the Incremental Specification of the Execution Semantics of Visual Languages

Paolo Bottoni
Dip. Informatica
Università La Sapienza
Rome, Italy
bottoni@di.uniroma1.it

Juan de Lara
Escuela Politécnica Superior
Universidad Autónoma
Madrid, Spain
jdelara@uam.es

Esther Guerra
Dep. Ingeniería Informática
Universidad Carlos III
Madrid, Spain
eguerra@inf.uc3m.es

Abstract

We present a new approach – based on graph transformation – to incremental specification of the operational (execution) semantics of visual languages. The approach combines editing rules with two meta-models: one to define the concrete syntax and one for the static semantics. We introduce the notion of action patterns*, defining basic actions (e.g. consuming or producing a token in transition-based semantics), in a way similar to graph transformation rules. The application of action patterns to a static semantics editing rule produces a meta-rule, to be paired with the firing of the corresponding syntactic rule to incrementally build an* execution rule*. An execution rule is thus tailored to any active element (e.g. a transition in a Petri net model) in the model. Examples from Petri nets, state automata and workflow languages illustrate these ideas .*

Keywords: Meta-Modelling, Visual Languages, Graph Transformation, Operational Semantics.

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 [9]. 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.

Two main approaches are currently employed to this aim, the grammatical and the meta-modelling one. The former exploits rewriting rules either to parse sentences and construct their interpretation [6, 16] or to define creation grammars, giving rise to syntax-directed editors. These allow some incrementality in the construction of abstract syntaxes, on which to build the semantic interpretation [16]. In some cases, different grammars are used for the incremental construction of a sentence and its subsequent parsing [1]. Triple Graph Grammars were proposed as a way to maintain forms of coordination between concrete and abstract syntax, favoring their incremental construction [20].

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. 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 [7, 4, 9, 14].

The definition of the dynamic semantics to be associated with a sentence – i.e. the type of domain-related process to be simulated with it – has in many cases to be carried out by hand and from scratch, as one has typically to consider different aspects, which may be arbitrarily complex. In particular, a designer has to define pre-conditions and triggers for a process transformation to take place, the types of resources it produces and consumes, and the functions for updating the associated values. Moreover, the dynamics to be modelled may refer not only to transformations of individual elements, but also to forms of coordination between them, to message exchange, diffusion of substances into a common environment, or to balances of forces [18].

In previous works, we introduced the notion of semantic variety, as expressed through a meta-model where the roles are identified that syntactic elements can play in a process [3], and followed in the line of the use of triple graph grammars as a way to couple syntactic and semantic roles [5]. In particular, we have proposed the use of triple patterns which allow the language designer to generate operational triple rules, simply starting from the definition of syntactic rules, once the correspondence between syntax elements and (static) semantic roles is established [8].

In this paper, we introduce the new notion of *action pattern* in order to generate execution semantics graph rewrit-

ing rules, whose application models some domain transformation. In particular, we show some basic patterns for the *token-holder* transition semantic variety, underlying the specification of several types of dynamics. In this variety, discrete transformations occur by removing tokens decorating holders, in a way which represents the holding of the transition preconditions, and decorating holders with tokens in a way which represents its post-conditions. Typical examples of languages with this type of (discrete) semantics 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 [18], or those describing chessboard games.

Paper organization. Section 2 discusses related work. Section 3 presents the meta-model for token-holder semantics and its specialization to transition-based languages. Section 4 presents a brief overview of graph transformation and introduces our approach for meta-rules. Section 5 introduces action patterns and Section 6 shows their application to rules defining the incremental construction of the static semantics. Finally, Section 7 discusses some applications, and conclusions are given in Section 8.

2. Related Work

Göttler describes a programming language as a triple formed of a syntax, a semantics, and a function specifying how the semantic model is built from the syntactic one [13]. He proposes meta-rules to modify either syntactic or semantic rules. In our case, meta-rules are associated with and triggered by syntactic rules, and are automatically generated from action patterns.

The use of action patterns to generate meta-rules is a form of meta-level manipulation of rules through rules. This has been exploited to define rule refinement through *rule* [19] and *subrule* [17] *morphisms*. In [19], a whole algebra of rules is defined based on rule morphisms, including operators for rule composition. Multiple matches for a rule into another would there give rise to different versions of the transformed rule, whereas in our case, the application of action patterns generates a single rule derived from the composition of the different matches.

Ermel and Bardohl [11] define an approach to animation in which the execution semantics is given in terms of transformations of configurations of the graph defining the process state, and analogous rules are defined for transforming an associated visualization. Rule morphisms then synchronize the application of rules in the process and visualization domains. The approach presented in this paper could be applied to visualization by considering the relation between static semantics and syntactic sentences (see [8]).

In Baresi and Pezzé's approach, 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 [2]. They also introduce a notion of notation family, to model commonalities in notations with slight differences in their interpretation. We remark that action patterns 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.

Taentzer uses amalgamation [21] to generate a specialized global execution transformation by considering all possible simultaneous matches for a set of rules, once the complete host graph has been produced. While the resulting execution semantics may not differ from our approach, the application of a parallel rule is not incremental (rules are generated on 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. We overcome some limitations of [21] e.g., checking in a Petri net whether all preconditions for firing a transition are satisfied is solved there 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). On the contrary, we produce specific execution rules for each transition, the action patterns for the case of Petri nets are more concise, and the framework is not tied in principle to any specific rewriting approach.

Multiset, rather than graph, rewriting is used in CIDER, a toolkit for the construction of smart diagram environments, supporting diagram editing, incremental construction of interpretations, execution of animations and diagram transformation [15]. CIDER supports different forms of behaviour, such as parallel or sequential transformations, and compositions of behaviours, as defined by control expressions. Transformations in CIDER can also be associated with constraints concerning the concrete graphical syntax, but need to be expressed through textual rewriting rules.

The notion of pattern is increasingly exploited in software and process engineering. In particular, Design Patterns [12] are defined as collaborations among different classes playing well defined roles. Patterns of execution have been studied in the modelling of workflow processes and a semantics for them has been given in the form of Coloured Petri Nets [22]. 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. Meta-Modelling for Syntax and Semantics

The definition of the syntax and static semantics of diagrammatic languages is based on the classes of Figure 1,

which shows two meta-models related through a correspondence meta-model. Such a *meta-model triple* [14] describes 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 in the lower part of Figure 1, semantic relations are expressed via *spatial relations* between *identifiable elements*. Different specialisations of these abstract classes define different families of visual languages [4], 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 modelling language belongs.

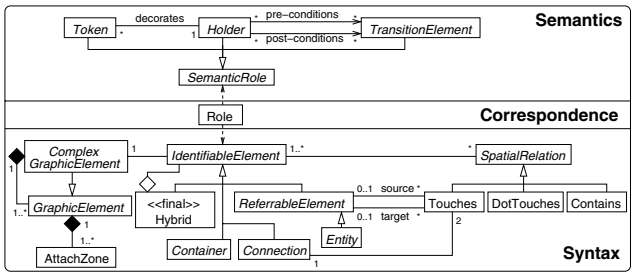

Figure 1. Meta-model Triple for Syntax and Semantics of Visual Languages.

The upper part of Figure 1 presents the basic classes for the *transition* semantic variety. In general, the notion of transition depends on that 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 a system instantaneous configuration, evolving under some well defined law. The possible evolutions at each step can be statically derived by the form of the diagram, or described externally. Internal descriptions of the admissible transformations rely on the presence of identifiable elements directly representing `TransitionElements`, with which `Holder` elements are associated as either `pre-` or `post-conditions`. Examples 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 in the specification of the deletion or creation of associations between `Token` and `Holder` elements. In particular, this execution semantics can be given through rules of type *before-after*, based on the differences in the way `Token` elements decorate `Holder` elements. For example, in grid-based languages such as Agentsheets, 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 moving of tokens from pre-condition holders to post-condition ones.

Figure 2 shows the definition of the concrete syntax and semantic roles for Petri nets. The significant spatial relations are refined (by means of a creation graph grammar) 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 `Places` and `Tokens`. Note that a `Place` can play both the role of an `Entity`, w.r.t the arcs referring to it, and that of a `Container`, w.r.t. the `Tokens` it holds.

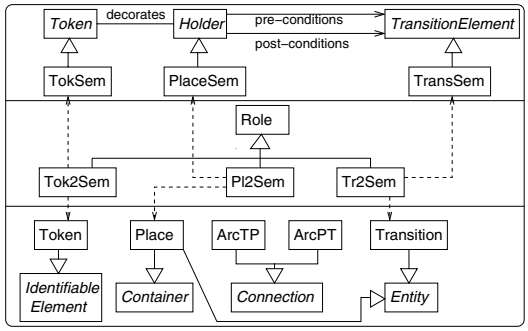

Figure 2. Meta-model Triple for Petri Nets.

According to the adopted syntax, the representation of the system dynamics can be directly supported by some form of canonical animation, in which instances of the `Token` abstract class 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 syntactic representations. For example, a Petri net can be represented by replacing transition boxes with hyperedges.

In [8], we used triple patterns to derive triple rules from rules acting only in the syntactic part. These modify synchronously the semantic model when the syntactic one is changed. Thus, in the rest of the paper, we concentrate on the semantic model (and no longer work with triple graphs) as, equivalently, we can apply our triple patterns to rules for the semantic model, and produce triple rules that synchronously modify the syntactic one. More precisely, they modify the corresponding abstract representation of the syntax, according to its meta-model, leaving to specific algorithms the management of the concrete layout.

4. Rules and Meta-Rules

In this section we give an informal, brief overview of the Double Pushout approach (DPO) to graph transformation. See [10] for a more extensive presentation.

Graph grammars are made of rules with a left and right hand side (LHS and RHS). When a rule is applied to a graph G (the *host graph*), an occurrence of the LHS (a matching morphism) has to be found in G, which can be then substituted by the rule's RHS. DPO uses category theory to model rules and derivations, and its theory has been lifted from graphs to (weak) adhesive HLR categories [10] (short (w)AHLR categories), based on a distinguished class $\mathcal{M}$ of monomorphisms. Examples of (w)AHLR categories are graphs, typed graphs, P/T nets and attributed typed graphs. Thus, not only graphs, but also objects in any (w)AHLR category $(\mathbf{C}, \mathcal{M})$ can be rewritten using DPO rules.

A DPO rule $L \xleftarrow{l} K \xrightarrow{r} R$ has three components (L, K and R), which are objects of a given (w)AHLR category. L contains the required elements to be found in the object to which the rule is applied. K (the gluing object) contains the elements to be preserved and R those that should replace the identified part in the object being rewritten. Roughly, $L-K$ are the elements deleted by the rule application, while $R - K$ are the elements to be added. Figure 3 shows a direct derivation diagram, where a production is applied to graph G yielding graph H, and (1) and (2) are pushouts in the given category. In particular, the category in the example is $\mathbf{Graph_{TG}}$ of typed graphs: an editing rule called *addPlaces* is applied to a host graph G, already containing a transition and a post-condition place for it. The rule adds one incoming and one outgoing place to the transition.

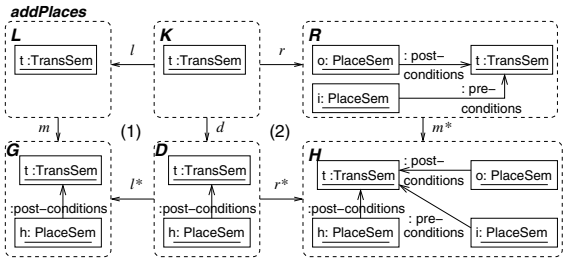

Figure 3. Editing Rule Derivation Example.

In our approach, one *execution rule* is created and associated with every transition or "active" element, to model its semantics. *Editing rules* are used to build the model, and they may be paired with one or more *meta-rules* to update the associated execution rule for each transition element in the editing rule. Such meta-rules are invoked each time a syntactic rule involving the corresponding transition element is triggered. These meta-rules modify the execution rule for the involved transition element, 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 $DPO(\mathbf{C}) = [\cdot \leftarrow \cdot \rightarrow \cdot, \mathbf{C}]$.

Figure 4 shows how the execution rule, named *Fire_t* associated with transition t in the context of the graph G in Figure 3 is transformed into *Fire_t"* to reflect the insertion of the two places in the transformed graph H. This is performed by the metarule shown in the upper part of Figure 4, to be associated with the editing rule *addPlaces* shown in Figure 3. The new execution rule specifies the removal of one token from the input holder and the insertion of one token into each output holder, as observable from the differences between the L and R components of rule *Fire_t"*.

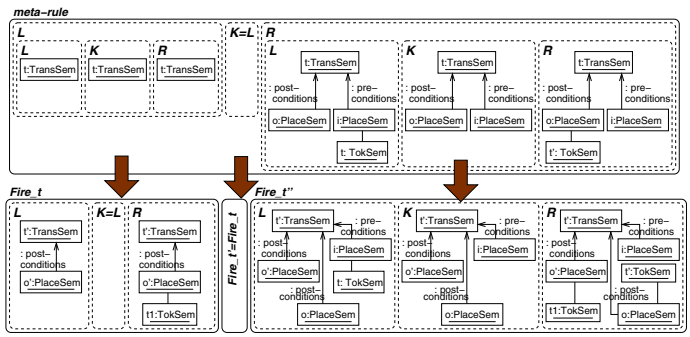

Figure 4. Meta-rule Rule Derivation Example.

5. Action Patterns

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*, similar to graph transformation rules [10], to describe semantics. We present a procedure to generate a meta-rule, starting from a set of patterns and an editing rule. This Section describes the notion of *action pattern*, and the next one the algorithm to obtain the meta-rule.

Let $TG = (N_T, E_T, s^T, t^T)$ be a type graph where N_T and E_T are sets of node and edge types, respectively and $s^T \colon E_T \rightarrow N_T$ and $t^T \colon E_T \rightarrow N_T$ define the source and target node types for each edge type. As in [7], we provide the type graph with node inheritance. A type graph with inheritance is a pair $TGI = (TG, I)$, where $I =$

(N_I, E_I, s^I, t^I) is a node inheritance graph, with $N_I = N_T$. That is, graph I has the same nodes as TG, but the edges of I are the inheritance relations.

Given a type graph with inheritance, the $clan$ of a node n is the set of all its children nodes (including itself). Formally $N_I \supseteq clan(n) = \{n' \in N_I | \exists \text{ path } n' \rightarrow^* n \text{ in } I\}$.

A *type system for patterns* over TGI is a construct $TSP = (TGI = (TG, I), tr, \sigma)$, where $tr \in N_T$ is a designated node type, for which the action patterns describe its semantics (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 $tr \in \sigma_{N_T}$. Elements in σ are needed for the expression of the operational semantics of the language (e.g. places, tokens and arcs).

An *action pattern* over TSP is a rule $ap : L^a \xleftarrow{l^a} K^a \xrightarrow{r^a} R^a$ such that $\bigcup_{n \in X^a_N} type(n) \subseteq \sigma_{N_T}$, for $X = \{L, K, R\}$, where X_N is the set of nodes of graph X. That is, an action pattern is a rule made of elements with type in σ. As in [7] for the case of rules, an action pattern may contain elements with abstract typing. In this case the pattern is called *abstract*. Elements with abstract type can get matched with elements of more concrete type (as in [7], similar to subtyping polymorphism). 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.

Figure 5 shows two abstract patterns describing a general transition semantics. The 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, the pattern *put* adds a token and an association with a post-condition holder. In both patterns a compact notation is used, showing elements L, K and R together in a single graph. The elements of $L - K$ (i.e. those that the pattern should delete) are marked as "$\{del\}$". Elements of $R - K$ (i.e. those that the pattern should add) are marked as "$\{new\}$". This notation will be used in the rest of the paper.

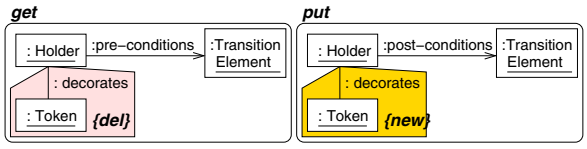

Figure 5. Action Patterns for Transition Based Semantics

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 meta-model triple in Figure 2). The type system for the patterns is given by the semantic meta-model (the upper one) in Figure 1, and the added subclasses by each particular language. In all cases, the distinguished element tr is $TransitionElement$. For the example of Petri nets, σ contains the classes in the semantic model of Figure 2. For some classes of Petri nets, where tokens with identities are used, one can introduce a *move* pattern, which does not remove or insert tokens, but only transfers the *decorates* association connecting the token from one holder to another.

6. Generation of Meta-Rules

In this section we present an algorithm that, given a set of action patterns and an editing rule, generates a meta-rule that updates an execution rule associated with a transition element. To provide intuition, we illustrate how the action patterns in Figure 5 can be applied to the editing rule of Figure 3 to obtain the meta-rule described in the upper part of Figure 4. As the editing rule adds a pre- and post- holder to an existing transition element, the associated meta-rule must update the execution rule by adding the semantics of an additional pre-holder and an additional post-holder. Hence, the meta-rule should identify the transition element 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. It is to be noted that the components of meta-rules are rules. The meta-rule's L component in Figure 4 is a rule with $L = K = R$.

In a situation as depicted on the left of Figure 6, the editing rule (and therefore the associated meta-rule) would have been fired twice for transition t. This would produce the execution rule shown to the right of Figure 6.

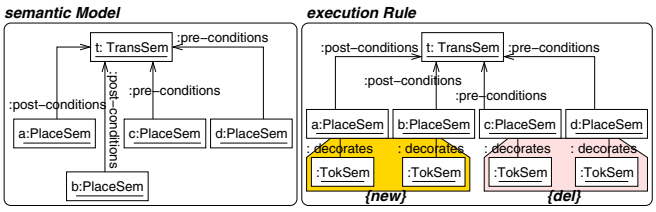

Figure 6. Semantic Model and Resulting Rule.

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

`Apply(AP:Set of ActionPattern, p:Semantic Rule, tsp:TypeSystemPattern): Set of Meta-rule`
Initialize the set of meta-rules, $MRS = \emptyset$.

$\forall t \in R|_{clan(tr)|_{\sigma_{N_T}}}$ ($R|_{clan(tr)|_{\sigma_{N_T}}}$ is the RHS of p, restricted to subtypes in σ_{N_T} of the designated node type *tr*):

- Initialize the meta-rule mr as follows $L_i^s = L'^s = K^s = K'^s = R^s = R'^s = K|_t$, where $K|_t$ is the kernel of the editing rule restricted to node t. Thus, the meta-rule becomes: $mr = (L^s \xleftarrow{id} K^s \xrightarrow{id} R^s) \Longrightarrow (L'^s \xleftarrow{id} K'^s \xrightarrow{id} R'^s)$ (we omit the meta-rule kernel K, as we work with non-deleting rules with $K = L$).
- Set $AP^c = \bigcup_{ap \in AP} conc(ap)$
- $\forall ap_j : L^a \xleftarrow{l^a} K^a \xrightarrow{r^a} R^a \in AP^c$:

1. Find all *injective* matches $N = \{n^i \colon K^a \to R\}$ from the kernel K^a of ap_j to the RHS R of the editing rule.
2. For $i = 0$ to $|N|$:
   (a) Calculate M_i as the pullback object of $kr \colon K'^s \to R$ and $n^i \colon K^a \to R$ as the left part of Figure 7 shows. Note that in the first iteration, $M_0 = K'^s = K|_t$.
   (b) Glue K'^s and $n^i(K^a)$ through M_i. This can be described using the categorical pushout construction as the right part of Figure 7 shows.

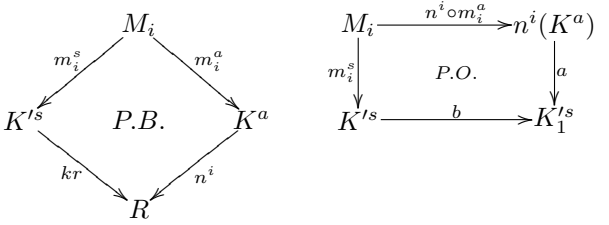

Figure 7. (Left) Obtaining M_i. (Right) Glueing K'^s and $n^i(K^a)$ through M_i.

   (c) Obtain the LHS (resp. RHS) of the execution rule (which is part of the meta-rule's RHS), by glueing L'^s (resp. R'^s) and L^a (resp. R^a) through M_i. These two processes can be described again as pushouts, as shown in the outer square of Figure 8 for the LHS (the RHS is built similarly).
   (d) Morphisms $l'^s \colon K'^s \to L'^s$ and $r'^s \colon K'^s \to R'^s$ are uniquely obtained from the pushout universal property: $K_1'^s$ is calculated to the right of Figure 7 as pushout object, and from Figure 8 we have $(c \circ l'^s) \circ m_i^s = (d \circ l^a) \circ n^i \circ m_i^a$. Hence, $\exists_1 l'^s \colon K_1'^s \to L_1'^s$, constructed as follows: for the elements in $K_1'^s$ coming from $n^i(K^a)$ (i.e. for those belonging to $a(n^i(K^a))$), we define $l'^s(x) = d \circ l^a \circ a^{-1}(x)$ [1]. For the elements in $b(K'^s)$, we define $l'^s(y) = c \circ l'^s \circ b^{-1}(y)$. A similar reasoning applies to the construction of r'^s.
   (e) Set $K'^s = K_1'^s$, $L'^s = L_1'^s$ and $R'^s = R_1'^s$.
- Update $MRS = MRS \uplus mr$

Return MRS.

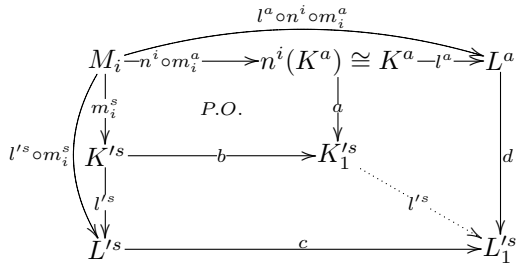

Figure 8. Obtaining the LHS of the Execution Rule (outer square is P.O.)

Figures 9 and 10 show some steps in the execution of the algorithm for patterns *get* and *put* and the rule of Figure 3 (the figures use abbreviated type names). The rule contains a single transition, thus one meta-rule is generated. First, we obtain all the concrete patterns for the Petri net language. The concretized *get* pattern is applied once. This pattern is like the one in Figure 5, but with elements of type `PlaceSem`, `TransSem` and `TokSem` instead of `Holder`, `TransitionElement` and `Token`. Figure 9 shows the construction of M_0, $K_1'^s$ and $L_1'^s$ (i.e. the RHS of the meta-rule except R). Similarly, Figure 10 shows the calculations for the application of concretized pattern *put*.

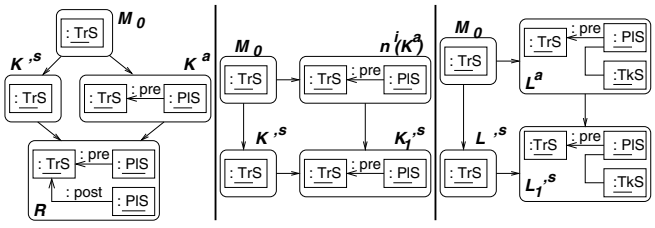

Figure 9. Applying Pattern *get*.

1 Note that a and b are monomorphisms, and therefore we can invert a in those elements belonging to $a(n^i(K^a))$

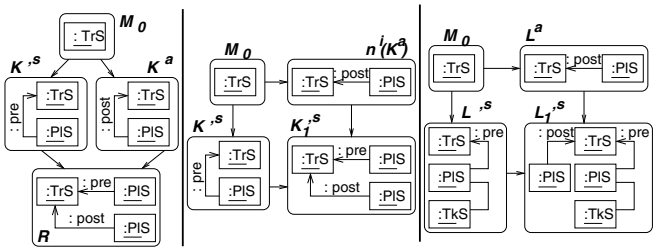

Figure 10. Applying Pattern *put*.

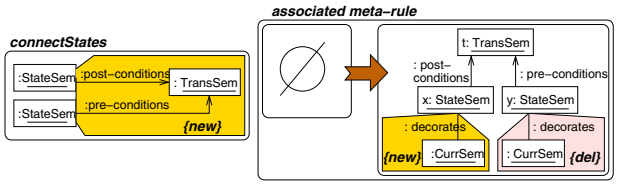

Figure 12. Editing Rule (left). Generated Meta-rule (right)

7. Applications

State Automata. Figure 11 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. one of them can receive a token, becoming the current state), while transitions are transition elements. For simplicity, we do not consider events in transitions.

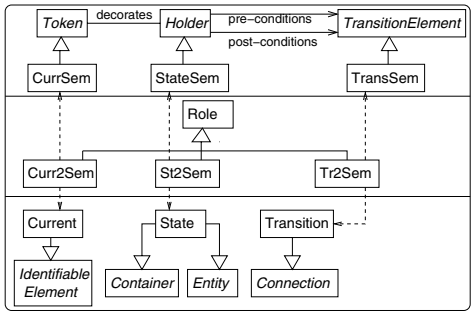

Figure 11. Meta-model for State Automata.

The action patterns *get* and *put*, shown in Figure 5 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 in this case is given by the semantic model in the upper part of Figure 11. Figure 12 shows an example editing rule and the generated meta-rule. In this case, as the transition element is created when connecting the two holders (i.e. states), the meta-rule creates the transition element in its RHS.

WorkFlow. Figure 13 shows an excerpt of a meta-model for a simple workflow language, in the style of [22]. 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. An example editing rule is shown to the upper-right corner of Figure 13. This rule adds a choice as a post-condition for the parallel block, while the latter becomes a pre-condition for the choice. On the other hand, 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 Figure 13 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 to include "Choice" in the determination of the context of the execution rules for "Parallel", but to avoid the generation of a meta-rule for the class "Choice" which does not need incremental semantics. Therefore, the type system TSP for this case includes in TG all types in the upper part of the meta-model triple of Figure 13, but excludes "Choice" from σ_{N_T}.

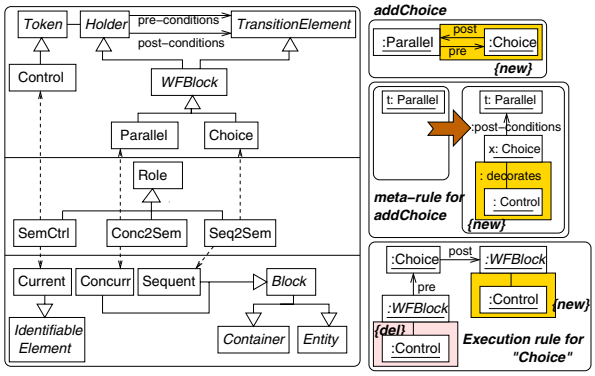

Figure 13. Meta-model for Workflow Language (left). Editing Rule and Generated Meta-rule; Exec. Rule for "Choice" (right).

8. Conclusions and Future Work

We have presented a new approach for the incremental specification of operational semantics for DSVLs, relying

on action patterns to describe the semantics of the active elements of a model. Patterns are applied to each editing rule manipulating active elements. Application of patterns produces meta-rules to be paired with the corresponding editing rules to incrementally build execution rules for the active elements (i.e. each time the editing rule is applied, the meta-rule is also applied, updating the execution rule). The key point is that meta-rules generate a tailored execution rule for each given active element, taking its context into precise account. The goal of this work is to reduce the information needed in order to define the operational semantics of a DSVL. This way, the DSVL designer only has to provide the action patterns, besides rules for the concrete syntax. Moreover, patterns can be reused for other editing environments with a different set of editing rules.

Many possible lines of development are open. For example, we have to consider attributes in graphs. Negative conditions (NACs) can be introduced at three different levels: to forbid the application of the action pattern; at the meta-rule level for creating or modifying NACs in the generated meta-rule; and at the execution rule level (i.e. the meta-rule would construct NACs in the execution rule). In addition, we are working to extending the algorithm in order to produce variations of the semantics, e.g. in case that not all tokens should be removed from every pre-condition holder. We are aware that, as currently defined, the patterns should be "smaller" than the editing rule to which they are applied. This is usually the case in editing environments; however the study of the opposite case is still an open problem. In principle, by considering partial matches from patterns to rules, one could devise ways to generate additional meta-rules with extended context. Finally, we are working in other semantic varieties, like the *communication* one [3].

Acknowledgements. Work sponsored by the EC with contract HPRN-CT-2002-00275, SegraVis, and the Spanish Ministry of Science and Education, projects MD2 (TIC200303654) and MOSAIC (TSI2005-08225-C07-06).

References

[1] R. Bardohl. A visual environment for visual languages. *Sci. Comput. Program.*, 44(2):181–203, 2002.

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

[3] 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*. IGP Press, to appear.

[4] 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.

[5] 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.

[6] G. Costagliola, V. Deufemia, and G. Polese. A framework for modeling and implementing visual notations with applications to software engineering. *ACM TOSEM*, 13(4):431–487, 2004.

[7] 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.

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

[9] 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.

[10] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

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

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

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

[14] E. Guerra and J. de Lara. Event-driven grammars: Relating abstract and concrete levels of visual languages. *Software and System Modeling*, to appear.

[15] A. R. Jansen, K. Marriott, and B. Meyer. CIDER: A component-based toolkit for creating smart diagram environments. In *Diagrams'04*, LNCS 2980, pages 415–419. Springer, 2004.

[16] M. Minas. VisualDiaGen - a tool for visually specifying and generating visual editors. In *AGTIVE'03*, pages 398–412, 2003.

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

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

[19] 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.

[20] A. Schürr. Specification of graph translators with triple graph grammars. In *Proc. WG94*, pages 151–163, 1994.

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

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