URL: http://www.elsevier.nl/locate/entcs/volume82.html 13 pages

Dynamic aspects of visual modelling languages ⋆

Paolo Bottoni [a]

[a] *Dipartimento di Informatica - Università di Roma "La Sapienza" - Italy*

Abstract

A large class of diagrammatic languages falls under the broad definition of "executable graphics", meaning that some transformational semantics can be devised for them. On the other hand, the definition of static aspects of visual languages often relies on some form of parsing or constructive process. We propose here an approach to the definition of visual languages syntax and semantics based on a notion of transition as production/consumption of resources. Transitions can be represented in forms which are intrinsic to the diagrams or external to them. A collection of abstract metamodels is presented to discuss the approach.

1 Introduction

The term "executable graphics" was introduced by Lakin [**?**], to emphasise that a graphical sentence can be considered as an executable specification of some process, if a suitable interpreter is provided. This definition was contrasted to that of visual programming languages, which was only focused on the purpose of the specification, i.e. program definition rather than communication in general. Such a notion lies today behind most visual modeling languages. In more general terms, independently of the purpose of the execution, we are interested in the existence of transformational semantics associated with graphical specifications, as is typical for visual modeling languages.

Transformational processes related to visual sentences can indeed be defined even for modelling languages not intended to specify processes but structures and configurations. The formal definition of diagrammatic languages is, in fact, often based on some intrinsic notion of process. In particular, syntactic definitions based on rewriting systems rely on some interpreter able to check the correctness of a visual sentence, or to generate correct visual sentences, with respect to the language specification. In the first case, we have a parsing

⋆ Partially supported by the EC under Research and Training Network SeGraVis and by Italian Ministry of Education.

process in which the rules in the language specification define which elements have to be present and how to reduce them towards obtaining a language axiom. In the second case, the interpreter executes the rules starting from some axiom to obtain a sentence which is guaranteed to be correct at the end of the generation process.

Examples of the first, *analytical*, approach are definitions through Constraint Multiset Grammars [**?**], Symbol-Relation Grammars [**?**], and Positional Grammars [**?**]. In these approaches visual elements are first represented symbolically and then the interpreter operates on these symbols, exploiting some suitable encoding of geometrical and topological properties and relations of the original visual elements. A survey of such approaches is in [**?**].

Examples of the second, *constructive*, approach are Shape Grammars [**?**], and Visual Conditional Attributed Rewriting Systems [**?**]. In these cases, the rules directly exploit the concrete visual elements, so that their application results in the transformation of a visual sentence.

Graph Transformations share properties of both approaches. Although originally employed to define visual languages in an analytical way, they rely on a mapping of the original sentence not into symbolic form, but into an abstract graphical form, mostly that of typed and attributed graphs. Hence, the analysis produces progressively simpler and more abstract graphs.

In many cases, diagrammatic languages are themselves specifications of processes. Again, we can distinguish two cases. In the first case, that we deem as *representational*, diagrams are used as an abstract representation of some behaviour, which can concretely involve any type of element. The process evolution may be mapped back to the animation of such diagrams. In the second case, *simulational*, the domain model is represented by visual elements whose visual behaviour is defined in visual terms (typically in the form of before-after rules involving the concrete visual elements), and the visual evolution is interpreted to portray aspects of the real evolution.

Examples of the representational approach are finite state machines, Petri nets, Statecharts. Examples of the simulational approach are Agentsheets [**?**], KidSim/Cocoa [**?**], Altaira [**?**]. The two approaches can be merged to obtain representational visual models specifying behaviours of simulational visual elements, as for example in [**?**]. Visual rewriting systems can also be used to specify visual behaviours. For example rewriting systems based on linear logic have been used to express the transformations defining the behaviour of finite state automata, as well as transformations defining visual inference processes [**?**,**?**]. Graph transformations are used in GenGEd to concurrently define behaviours and their representations as animations of visual elements associated with graph nodes [**?**]. General purpose visual programming languages, such as Pictorial Janus [**?**] or Vex [**?**], are beyond the scope of this paper, but they appear to share aspects of representational languages.

In general, configurations of (symbolic or visual, concrete or abstract) elements are transformed into some other configuration. The content of the

transformation is in turn often defined in visual terms, while its execution is usually specified by an abstract interpreter, often defined in algorithmic terms. In this paper we aim at defining a general model of visual transformations as based on an abstract notion of transition, seen as production/consumption of resources, and on an abstract notion of transition step, as defined by some application policy. In particular, whereas models of specifications of visual transformations specify at the same time the form of the rules and the way in which they have to be applied, we argue here that a separate definition of these two aspects allow a greater flexibility, and in particular reuse of specifications under different application policies.

The rest of the paper develops as follows. After revising some related work in Section 2, we propose a metamodel for diagrammatic languages in Section 3, regarding a visual modeling language as composed of visual elements in some significant spatial relations. This metamodel is then related to a metamodel for visual specification of transitions, and to a more general metamodel for transformations in Section 4. Finally, Section 5 discusses visual representations of processes and the management of visual interaction, before conclusions are given in Section 6.

2 Related work

Several authors have studied abstract definitions of visual languages as a basis to define formal semantics on them, or for the construction of visual tools.

Erwig introduced the notion of abstract syntax for visual languages, adopting a graph-based approach [**?**], in contrast with tree-based abstract syntaxes typical of textual programming languages. The denotational semantics of a diagram is then constructed from such a graph. In this paper, we are more interested in the transformations which occur on the graphics, leaving their mapping to some external semantics to a separate interpretation process.

On the other hand, approaches based on graph transformations usually exploit a distinction between a low-level and a high-level interpretation, possibly occurring on distinct graphs [**?**]. In the approach proposed here, we assume the existence of suitable realisations of geometry so that we can omit considering the low-level interpretation, and we concentrate on the definition of constraints relating the different components of a visual sentence. A recent proposal refers to *category theory* to characterise families of connection-based languages in terms of morphisms among elements [**?**]. Category theory has also been used to specify the semantics of component-based visual programs, in which connections define some communications among them [**?**]. Classes of languages were defined according to an Entity-Relationship approach in [**?**].

Metamodel approaches are gaining interest in the visual language community, following their success in defining the visual languages in UML. In particular, the UML approach combines diagrammatic sentences (usually class diagrams) and textual constraints for the definition of the semantics of visual

languages. A metamodel approach is implicit in most generators of diagrammatic editors, in which at least an abstract notion of graphical object has to be defined. Most generators are based on the translation of some formal syntax into a set of procedures controlling user interaction, and on constructing the semantic interpretation of the diagram in a parsing process. Examples of such tools, where the formal syntax is some variant of graph rewriting, are Diagen [?] (based on hypergraph rewriting) and GenGEd [?] (in which constraints are used to define lexical elements, and two separate graph rewriting systems are used, one to guide diagram construction, and one to define the parsing process). In MetaBuilder [?], a designer defines a new visual language by drawing its metamodel class diagram, from which an editor is automatically generated, but the diagram transformational semantics is not considered.

Moses [?] follows [?] in working on an abstract syntax description of visual sentences through attributed graphs. From this specification a syntax-checker is defined. In the Moses environment a user can define a sentence, with the syntax-checker operating in the background. Semantics is dealt with by producing specific interpreters in the form of Abstract State Machines for any given visual language. We will see in Section 5 how modeling transformation systems, visual or not, in terms of production and consumption of resources can support the automatic generation of syntax-directed editors.

Metamodels are used in the AToM^3 environment [?] as a general form of abstract syntax. Meta-meta-models generate meta-models defining the type of systems to be simulated. Finally, a model defines a specific instance of system. All models are expressed in some visual formalism and model transformations are defined through graph grammars. Component interconnection is solved by mapping different components to a common formalism.

The research described in this paper has important similarities with the GRACE effort towards a unified view of graph transformations [?]. GRACE strives to achieve approach independence via an axiomatic definition of graph transformation, so as to be able to combine the semantics of different transformation systems. Distributed systems are thus realised as composition of modules via some import and export interfaces. In our approach, we regard multiset, rather than graph, rewriting as the basis for modelling transformations, and we stress the possibility of having reconfigurable interpreters managing flexible policies for rule application.

The EU working group APPLIGRAPH is leading an effort to define common exchange formats for graphs and graphs transformations [?]. In this case, two logical models in the form of UML class diagrams are the basis for two DTDs, so that XML can be used as an exchange format. We will discuss later similarities and differences between our proposals and those in [?].

3 Visual sentences

We base our proposal on a view of of visual sentences as composed of *visual resources*, expressable as terms of the form $t(s, x_1, \ldots, x_n)$, where t is a *type symbol* from an alphabet Σ, s is a *graphical structure* and each x_i is the value of some attribute. Each term type t is characterised by a *graphical type* of the same name, to which the structure s can belong, and by a set of attributes. We call $K(\Sigma)$ the set of visual terms constructed from the elements of Σ and $\mathcal{D}(K(\Sigma))$ the set of diagrams constructed with terms from $K(\Sigma)$.

In particular, refining the metamodel proposed in [**?**], we consider meta-meta-models defining families of visual modelling languages, where each modelling language provides a metamodel for the definition of models of systems. For brevity, we will use the term metamodel in all cases. We do not discuss here the structural aspects of the classifiers in the class diagrams defining the abstract syntax of each family, except for the presence of aggregations. Figure 1 shows the metamodel to which diagrammatic languages conform, meant as languages whose interpretation depends on the identification of significant *SpatialRelation*s among *IdentifiableElement*s. A set of elements can be involved in more than one spatial relation. For example, if the relations `leftOf` and `above` are each defined by partitioning the plane in two semiplanes based on the position of the source element, then any two elements not exactly aligned will participate in both relations. The attributes of a visual term are defined by the corresponding identifiable element. An identifiable element is associated with a *ComplexGraphicalElement* which manages its geometrical aspects, thus defining the appearance of its graphical structure. A complex graphic element can in turn be composed of several graphical elements at a lower level of abstraction. For each graphical element a collection of *AttachZone*s is defined, whose concrete realisations are of type *Area*, *Border*, or *Dot*. The existence of a spatial relation among elements must be decidable on the basis of the definition of the *isAttached* operations in the attach zones of the graphical elements.

Apart for the realisations of *AttachZone*, all classifiers in Figure 1 represent abstract classes except for the case of *AttachZone*, which is an interface, and of *Polyline* and *Icon*, which are here concrete classes as defined in the UML specification document, while *Shape* is left abstract as is common in object-oriented hierarchies for graphical toolkits.

An important family of diagrammatic languages is that of *connection-based* languages, whose metamodel is presented in Figure 2. In this family, identifiable elements can be either *ReferrableElement*s or *Connection*s. The graphical element associated with a connection is of type *Polyline* (this would be expressed by an OCL constraint), while *Touches* is the only type of significant spatial relation. According to the type of connection, i.e. if *isDirected()* or *isHyper()* return *true*, the *elements* association end is specialised in *source*, *target*, or *members*, with suitable multiplicities. Some connections may in

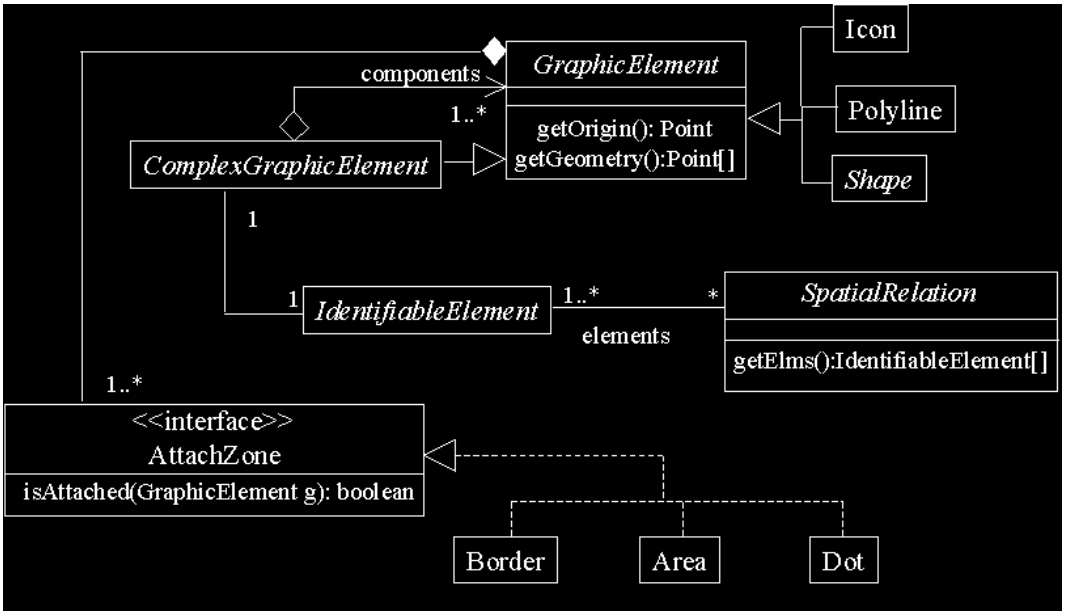

Fig. 1. The basic model for diagrammatic languages

turn be *ReferrableConnection*s, while an *Entity* is a referrable element which cannot act as a connection. This family can be further specialised. For example, *graph-like* languages have graphical structures associated with entities with their whole borders as attach zones, while in *plex* languages the graphical structures have a fixed set of attaching dots. The logical model for graphs presented in [?] can be seen as a refinement of the part of our abstract syntax related to identifiable elements (i.e. entities and connections, there expressed as nodes and edges), without considering the characteristics of graphical elements, which are therefore excluded from the XML documents.

4 Specifying execution

In general, the notion of transition is connected to the identification of a notion of system *state*, significantly changed by the occurrence of the transition.

We restrict ourselves to considering identifiable elements in a *Diagram* which are *SemanticElement*s, i.e. those visual elements through which semantics is defined, and will abstract from non significant marks, noisy elements, or so on which may occasionally be present in a concrete diagram. Some types of identifiable elements, such as annotations, borders, and so on, are also present in a diagram and constitute the so-called *ParatextElement*s, useful for the comprehension of the overall meaning of the diagram, but not involved in determining the characteristics of the transitions. For the case of visual transformations, it is important to distinguish between what may change and what has to remain constant along a transformation, or possibly be subject to a restricted set of transformations. Typically, in [?] a *frame* has been defined as that set of visual terms whose structures maintain the same

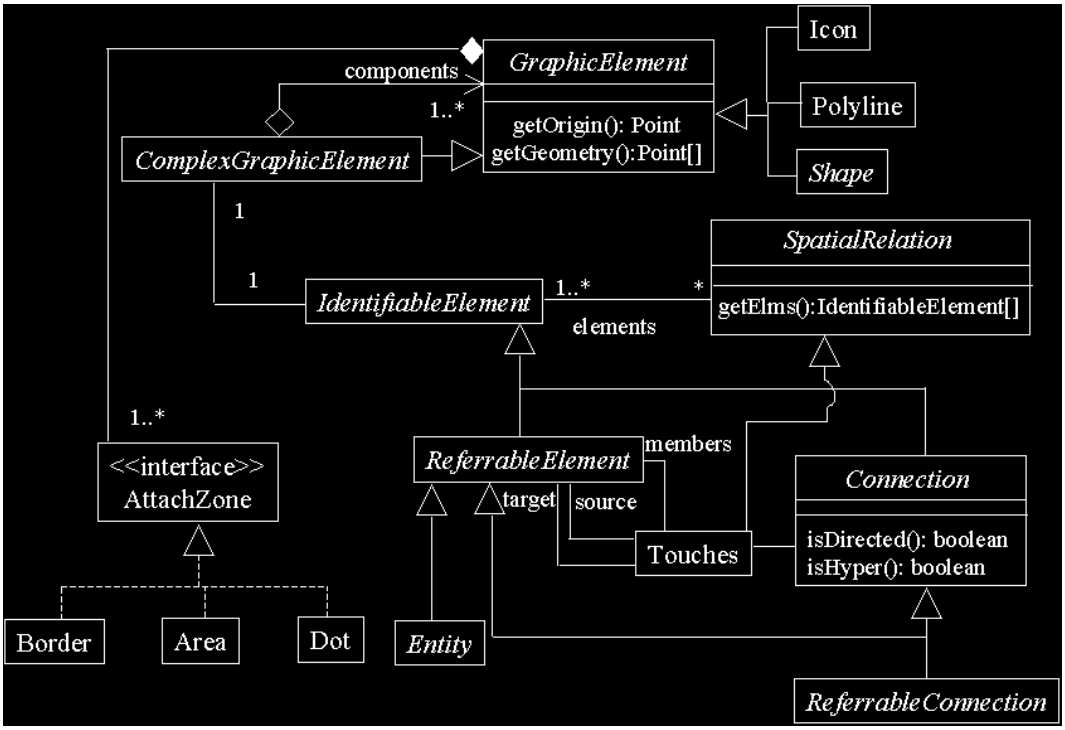

Fig. 2. The model for connection-based languages

geometrical properties along a transformation, in particular their coordinates, possibly varying the colour of the pixels. Such semantic elements provide a context in which an observer can understand the content of the transformation. Some elements, called *ContextSupportElement*s are typically devoted to this role. A *VisualConfiguration* is then a projection of a diagram, involving only those semantic elements which can be subject to change. We call such elements *ConfigurationSupportElement*s. We use the term configuration, instead of state, to underline its distributed aspect and the fact that the overall state descends also from the overall structure of the relations among the elements. We can now explore some typical forms for specifying behaviours through visual notations. It appears that three main types of specification can be identified.

In a first case, diagrammatic notations are used to specify transformations following some prototypical model such as finite state automata, dataflow diagrams and their derivations, or Petri nets. These models use a diagram to implicitly define the content of the transformation (which can be represented as diagram animation), while adopting an explicit representation of the transition elements. The definition of the actual operational semantics for the transition is delegated to an external definition, in the form of an algorithm or of a rewriting relation. These models have canonical visual representations, so that one can for example identify the mathematical definition of a finite state automaton with any one of the isomorphic state-arrow graph representations of the automaton. These visual representations constitute specific visual

languages, typically, in the family of connection-based languages. The explicit representation of the transitions in the diagram is realised by the presence of special types of semantic elements, called *Transition*s. As they remain constant in these types of models, we consider them as special cases of context support elements. The representation of the system dynamics is directly supported by some form of canonical animation. To this end, we consider a special type of configuration support elements, called *Token*s, which may appear or disappear, or move along the transition from a *Holder* to another. As holders remain constant in the animation, and provide a basis for interpreting the occurrence of a transition, we also consider them as context support elements. A visual configuration is actually determined by how tokens *decorate* holders. Figure 3 shows the metamodel resulting from the analysis above, and defines the family of visual languages using explicit representations of transitions and supporting execution as animation of the specification. The connection with the concrete visual level (as described in the metamodel of Figure 1) is realised by mapping semantic elements to identifiable elements or their properties. For example, a token need not be a separate graphic element, but its presence or absence can be represented by some special appearance of the graphic element associated with a holder, e.g. colouring of a state node is often used to indicate the current state. Conversely, the presence of a token in one of a given set of holders could be represented by associating different icons with the token. While, as said before, the representation of transitions in this family of languages usually involves the use of connections (as in Finite State Automata), it is also possible that transitions be represented by referrable elements as happens in Petri nets. In any case, the *Touches* spatial relation is used to associate some pre- and post-conditions of a transition with the transition itself. Moreover, attributes of configuration support elements relevant to defining their status are mapped into visual attributes of identifiable elements. The same may happen for transitions, if one wants to represent some state the transition can be in, or record the story of its activations. Such representations are in general used only for the animation, while the static definition of the specification does not usually resort to such artefacts.

The second family groups those visual specifications which express transformations as before-after visual rules, following the pioneering works of BIT-PICT [**?**], Agentsheets [**?**] or KidSim (today: Stagecast) [**?**]. These are based on a simple operational semantics, which can be described as: "'remove a subdiagram L which is an occurrence of the left-hand side from the current diagram and substitute it with a subdiagram R which is an instance of the right-hand side generated on the basis of L". The mapping $f : \mathcal{D}(K(\Sigma)) \rightarrow \mathcal{D}(K(\Sigma))$ such that $R = f(L)$, can be defined in visual terms or not. However, such languages do not need to have a canonical application policy, even though implementations usually support simple policies, e.g. sequential or concurrent.

Finally, visual rules can be associated with sophisticated application poli-

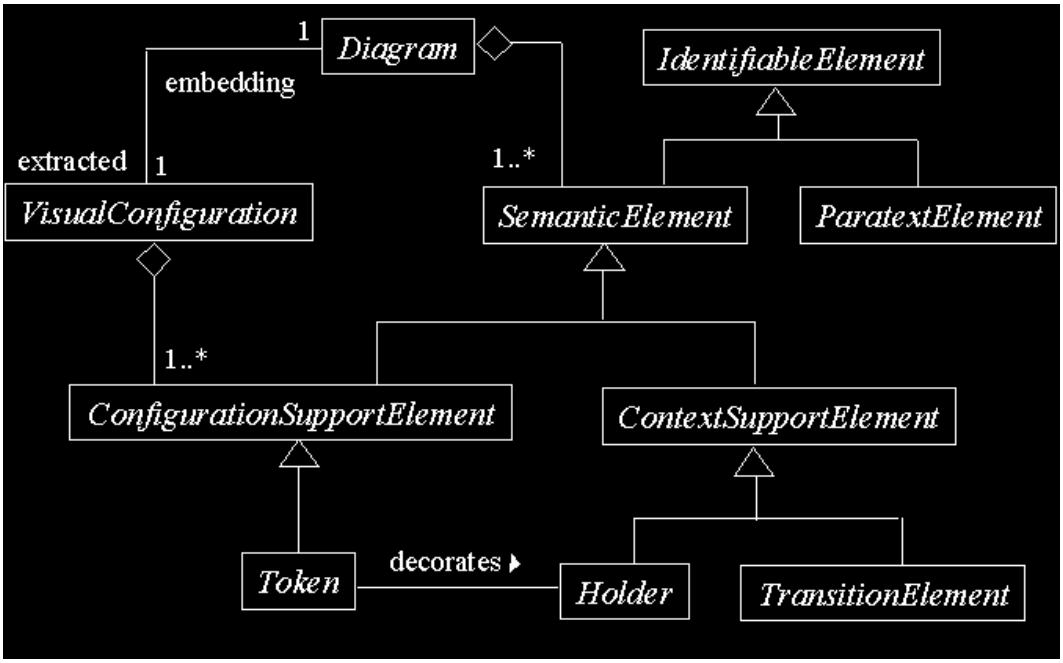

Fig. 3. The model for visual languages using explicit representations of transitions.

cies or embedding mechanisms, as in graph grammars (for a survey, see [?]). In this case, however, rules operate on an abstract representation of the diagram, rather than on the actual diagram. Transformations can be mapped back to the actual diagram, generally by way of programmed routines [?], or layered transformations [?] (for a survey on applications see [?]).

In general, all families of visual specifications rely on a general abstract model of what a visual transformation is. We consider that a *RuleSet* is a collection of *TransformationSpecification*s, each composed of a set of *Preconditions* and a *Postcondition* associated via some *mapping*. A transformation specification can involve some additional *MetaConstraints*, as for example definition of priorities, or grouping of rules. The definition of pre- and post-conditions can be reduced to the specification of the *Patterns* for the *Resources* which are needed in a *pre-matched Configuration* for the transition to be applied and of those that are defined as present in the *post-matched* configuration after application. The *Matching* abstract class specifies the characteristics of the required matching (e.g. injective). Some preconditions of a rule (but not all) may appear in negated form, implying that they must not be matched by the configuration for the transition to be applied. Some *restriction* can be applied between the negative and the positive pre-conditions. The mapping to the post conditions originates only from the positive preconditions. All patterns are typed. A *Type* is associated, via *lists*, with the *ordered* sequence of *Attributes* describing the features of the resource. These attributes can be referred to in the pattern through variables and literals, or be left unmentioned. In any case, the attributes considered in the pattern are a subset of

those defined by the type. Hence, a pattern actually defines a set of resources complying with the values of the attributes instantiated in it. An *Alphabet-Symbol* is a pattern which mentions all the attributes defined by the type as variables. Each resource is an instance of an alphabet symbol with a literal value for each attribute. The specification can also require the performance of some *Activity*, which may use as parameters the attributes mentioned in the patterns in the pre- or post-conditions. Typically, activities in a pre-condition are meant to evaluate constraints and to be free from side effects. Activities in a post-condition are usually meant to produce values for instantiating the patterns mentioned in the post-condition, but can also start some external process, modifying the environment in which the system is immersed.

A *TransformationStep* applies some transformation specification according to some *Policy*, and transforms a *source* configuration into a *target* one, according to the meta constraints associated with the selected specifications. The source configuration must provide a pre-match for each transformation specification *activated* in the transformation specification. In a similar way, the target configuration must provide a post-match for the same specifications, where the pre and post-matches are consistent with the mapping between pre- and post-conditions. The abstract syntax for the resulting metamodel is presented in Figure 4. Again, the model for graph transformation systems of [?] can be seen as a specialisation to the case of graphs of this general model. In particular, it makes the notion of *embedding* explicit, which can be expressed in our model through pre- or post-conditions, constraints, or activities.

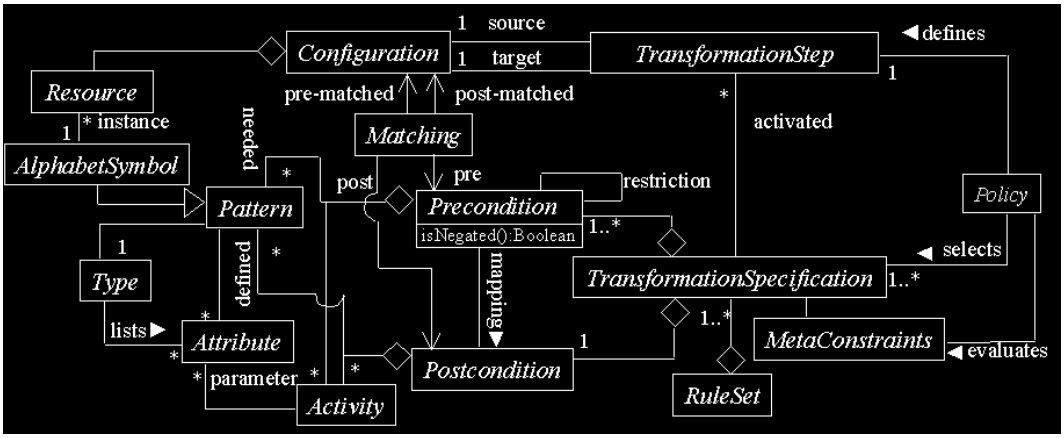

Fig. 4. A general model for specifications of transformation processes

5 Relating processes and visual representations

While the abstract syntax of Figure 4 provides a general model for any kind of (discrete) transformation system, it can be immediately related with the meta-model discussed with reference to Figure 3, with the needed simplifications in the case that transitions are not explicitly mentioned. Hence, semantic ele-

ments can be seen as specific types of resources, so that we eventually employ graphic elements or their properties to represent resources. Each transformation specification must be expressed in terms of the effect it has on the graphic elements present in a diagram. Hence, a general mechanism can be devised to relate abstract and visual transformation processes. If we consider identifiable elements as visual resources, we can provide a mapping between the alphabet of resources in the modelling languages and the alphabet of identifiable elements in their visual counterparts. It is then possible to define arbitrary chains of such mappings, so that even modelling languages with canonical visual representations can be represented with any type of graphic elements. Conversely, visual specifications can be transformed into abstract definitions of transitions, provided one can map in a non ambiguous way the identifiable elements in the visual specification to resources of the model.

Transition specifications are here considered independent of the application policy governing their concrete activation. In this way, even transition specifications which have an associated canonical model of execution can be applied with different models. For example, one could use Petri nets with the usual true concurrency semantics, or prescribe a sequential application policy where only one transition per step can be applied. Note that this would require the transformation of the original Petri net into one with an additional place which is a pre-condition for all the transitions in the net and in which a token is forced to be put back by each transition. One can even prescribe a maximally concurrent semantics, where all transitions that *may* occur without conflicts *must* occur. Such a type of behaviour would be modelled with a Petri net of exponential complexity derived from the original one, as one should take into account all possible simultaneous firings.

The application policy can also accommodate particular forms of embedding rules constraining the application of a transformation, or be based on an interpreter for control expressions, possibly prescribing the transactional application of rules, as occurs for transformation units in graph transformations [**?**]. For example, the request of the SPO approach, that edges connected with removed nodes be removed as well, can be realised by transforming the original SPO rule $r : L \to R$ into a a transformation unit $T(r)$ which defines a step as resulting by a rule expression prescribing the following:

(i) mark the nodes and edges to be removed as prescribed in L (this rule may use arbitrary matching);

(ii) as long as possible do remove an edge connected to a marked node;

(iii) remove the marked edges and the marked nodes if there is no edge attached to any of them and insert the graph specified in R, respecting the mapping r.

The views presented so far are also relevant to the management of visual interaction. In particular, interactive systems based on the paradigm of the *separation of concerns* between model, view, and interaction control, can be

considered as the pairing of two dynamic systems through a causal connection.

The *Control level* system is defined by a transition system where, apart from the resources defining its configuration, additional resources can be introduced by the user by performing specific well-defined types of *action*.

The *Visual level* system maintains a collection of (visual) resources which define its configuration. Transformations in the configuration occur according to transition rules which can be fired by the control system, so as to reflect modifications in the underlying *Model level* system.

In particular, in the design of syntax-directed visual editors, visual rules defining the valid sentences in the visual language can be considered as the specification of transformations in the visual configuration corresponding to the current state in the construction of the visual sentence. On the other hand, according to the style of interaction, such rules define standard sequences of transformation steps in the control system to be performed in order for a transformation at the visual level to occur. In [**?**,**?**] patterns for the automatic construction of control systems for a family of visual rewriting systems are presented. In [**?**], it is shown how the pair (Control level, Visual level) can be automatically generated starting from such visual rewriting systems. A syntax-directed editor results therefore from the pairing of two communicating transition machines. One machine exploits a sequential policy to realise the control level in the form of transitions of a conditioned transition network, where each state represents a selection of visual resources and identifies two sets of user actions: enabled and disabled ones [**?**]. The second machine manages the evolution of the visual sentence according to the original set of rules. In this case different policies can be adopted, for instance if some sequence of rule applications specified by a transformation unit must occur. In any case, transitions in this second machine are triggered by resources produced by the control machine in correspondence of some well defined transitions. The communication between the two machines is based on import/export declarations of types of resources.

Figure 5 describes the conceptual architecture for a generator of syntax directed editors. From a *LanguageSpecification* composed of an *Alphabet* and a *RuleSet*, a *Translator* creates two rule sets: a, defining the behaviour of a control automaton governing the interaction with the user, and s, governing the evolution of the diagram, i.e. the creation, deletion and modification of the identifiable elements in it. The two specifications define the behaviours of the *aut* and *sent Machine*s, implementing the activation policies and maintaining the configurations of the two systems. The user actions producing transitions in *aut* are defined as `selection`, `deselection`, `generation` and `deletion` of alphabet elements. The actual generation or deletion of a diagram element is ultimately determined by the rules defining the transitions in *sent*. The causal connection between the interactive level and the diagram level is managed by an *Executor*, which guarantees the communication between the two machines. Some *Observer*s reflect the evolution of the two machines back to the user

or communicate it to other components of an interactive system. The greyed components in Figure 5 describe the fixed components of the architecture, while the others depend on the specific transformation set.

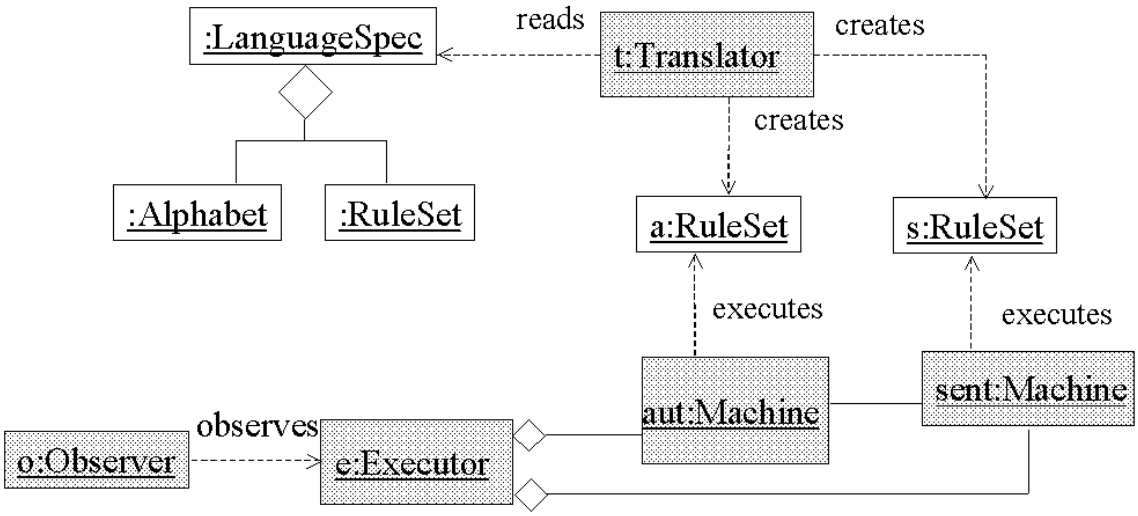

Fig. 5. Syntax directed editors seen as paired dynamic systems

6 Conclusions

We have discussed a view of dynamics in visual sentences which regards diagram transformations as production and consumption of identifiable elements in a visual configuration. Such elements are ultimately conceived as graphical resources. This approach relies on the availability of generic interpreters for transformation specifications, which can be applied according to arbitrary policies. The interpreters can thus be dynamically configured by changing independently the rules to be applied or the application policies. This favours the coordinated management of heterogeneous notations as well as the automatic generation of dynamic systems from specifications. The approach can be uniformly applied to the management of transition systems defined by visual specifications as well as to the generation of visual sentences via syntax-directed editors, or to define visualisations of parsing processes in which the reduction actions are modelled as the substitution of groups of identifiable elements with graphical representations of non terminals.

Acknowledgements
I thank the members of the Pictorial Computing Laboratory at the University of Rome: S. Levialdi, M. De Marsico, D. Ventriglia, and P. Di Tommaso for several valuable discussions, and the latter two for contributing to the implementation. The anonymous UNIGRA referees are also thanked.