Event-Driven Grammars: Towards the Integration of Meta-Modelling and Graph Transformation

Esther Guerra and Juan de Lara

Escuela Politécnica Superior
Ingeniería Informática
Universidad Autónoma de Madrid
`Esther.Guerra_Sanchez@ii.uam.es, Juan.Lara@ii.uam.es`

Abstract. In this work we introduce *event-driven grammars*, a kind of graph grammars that are especially suited for visual modelling environments generated by meta-modelling. Rules in these grammars may be triggered by user actions (such as creating, editing or connecting elements) and in its turn may trigger other user-interface events. Its combination with (non-monotonic) triple graph grammars allows constructing and checking the consistency of the abstract syntax graph while the user is building the concrete syntax model. As an example of these concepts, we show the definition of a modelling environment for UML sequence diagrams, together with *event-driven grammars* for the construction of the abstract syntax representation and consistency checking.

Keywords: Graph Grammars, Meta-Modelling, Visual Languages, Consistency, UML.

1 Introduction

Traditionally, visual modelling tools have been generated from descriptions of the Visual Language (VL) given either in the form of a *graph grammar* [2] or as a *meta-model* [6]. In the former approach, one has to construct either a *creation* or a *parsing* grammar. The first kind of grammar gives rise to *syntax directed* environments, where each rule represents a possible user action (the user selects the rule to be applied). The second kind of grammars (for *parsing*) tries to reduce the model into an initial symbol in order to verify its correctness, and results in more *free editing* environments. Both kinds of grammars are indeed encodings of a *procedure* to check the validity of a model.

In the *meta-modelling* approach, the VL is defined by building a meta-model. This is a kind of type graph with multiplicities and other – possibly textual – constraints. Most of the times, the concrete syntax is given by assigning graphical appearances to both classes and relationships in the meta-model [6]. For example, in the AToM[3] tool, this is done by means of a special attribute that both classes and relationships have. In this approach the relationship between concrete (the appearances) and abstract syntax (the meta-model concepts) is one-to-one. The meta-modelling environment has to check that the model built by the user is a correct instance of the meta-model. This is done by finding a typing morphism between model and meta-model, and by checking the

defined constraints on the model. In any case, whereas the graph-grammar approach is more *procedural*, the meta-modelling approach is more *declarative*.

In this paper we present a novel approach that combines the meta-modelling and the graph grammar approaches for VLs definition. To overcome the restriction of a one-to-one mapping between abstract and concrete syntaxes, we define separate meta-models for both kind of syntaxes. In a general case, both kinds of models can be very different. For example, in the definition of UML class diagrams [12], the meta-model defines concepts *Association* and *AssociationEnd* which are graphically represented together as a single line. In general, one can have abstract syntax concepts which are not represented at all, represented with a number of concrete syntax elements, and finally, concrete syntax elements without an abstract syntax representation are also possible. To maintain the correspondence between abstract and concrete syntax elements, we create a *correspondence meta-model* whose nodes have pairs of morphisms to elements of the concrete and abstract meta-models.

The concrete syntax part works in the same way as in the pure meta-modelling approach, but we define (non-monotonic) triple graph grammar rules [11] to build the abstract syntax model, and check the consistency of both kinds of models. The novelty is that we explicitly represent the user interface events in the concrete syntax part of the rules (creating, editing, connecting, moving, etc.) Events can be attached to the concrete syntax elements to which they are directed. In this way, rules may be triggered by user events, so we can use graph grammar rules in a free editing system. Additionally, we take advantage in the rules of the inheritance structure defined in the meta-model, and allow the definition of *abstract (triple) rules* [3]. These have abstract nodes (instances of abstract classes in the meta-model) in the LHS or RHS. These rules are equivalent to a number of *concrete rules* obtained from the valid substitutions of the abstract nodes by concrete ones (instances of the derived classes in the meta-model). We extend this concept to allow refinement of relationships.

As a proof-of-concept, we present a non-trivial example, in which we define the concrete and abstract syntax of sequence diagrams, define a grammar to maintain the consistency of both syntaxes, and define additional rules to check the consistency of the sequence diagram against existing class diagrams.

2   Meta-modelling in AToM^3

AToM^3 [6] is a meta-modelling tool that was developed in collaboration with Hans Vangheluwe from McGill University. The tool allows the definition of VLs by means of meta-modelling and model manipulation by means of graph transformation rules. The meta-modelling architecture is linear, and a *strict* meta-modelling approach is followed, where each element of the meta-modelling level n is an instance of exactly one element of the level $n + 1$ [1].

Figure 1 shows an example with three meta-modelling levels. The upper part shows a meta-metamodel for UML class diagrams, very similar to a subset of the core package of the UML 1.5 standard specification. The main difference is that *Associations* can also be refined, and that the types of attributes are specific AToM^3 types. Some of the concepts in this meta-metamodel are *Power types* [10], whose instances at the lower
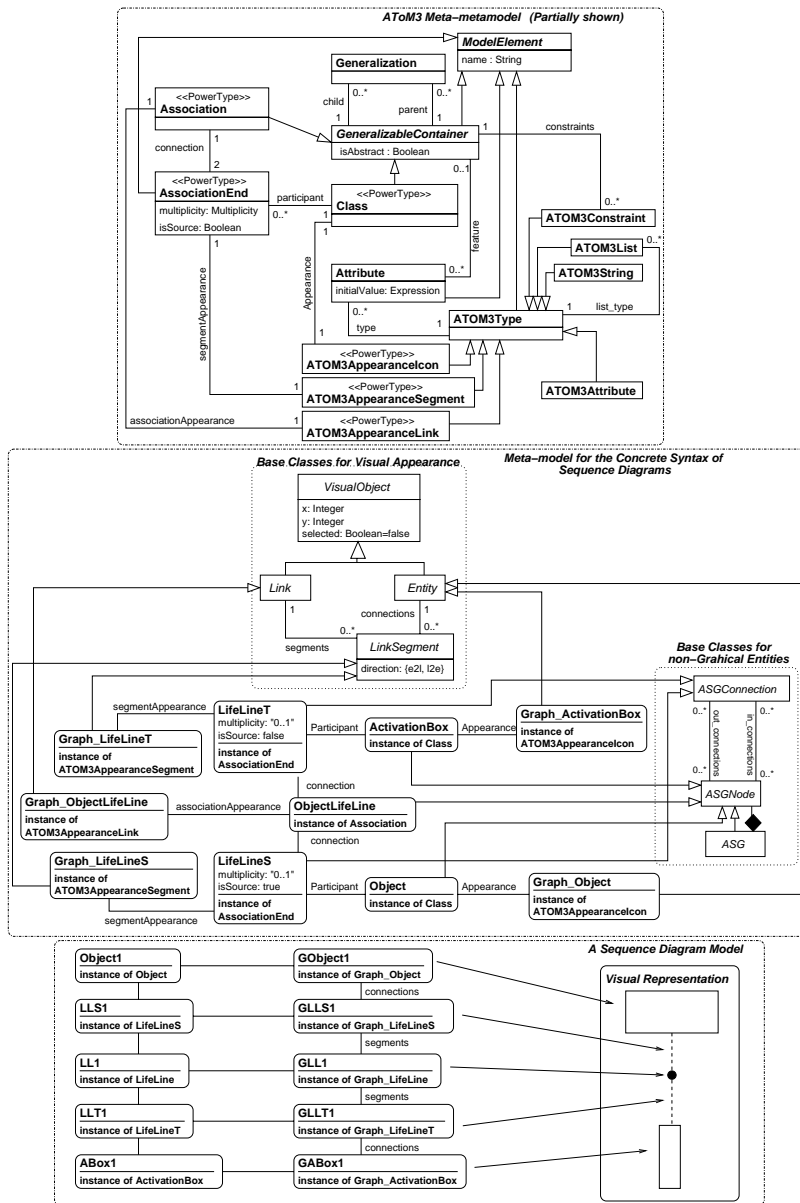
Fig. 1. Meta-modelling levels in AToM[3]

meta-level inherit from a common class. This is the case of *Class*, *Association* and *AssociationEnd*, whose instances inherit from *ASGNode* and *ASGConnection*. Classes *ATOM3AppearanceIcon*, *ATOM3AppearanceSegment* and *ATOM3AppearanceLink* are special types, which provide the graphical appearance of classes, association ends and

associations. They are also *Power types*, as their instances inherit from abstract classes *Entity*, *LinkSegments* and *Link*. The user can define the visual appearance of these instances with a graphical editor. Instances of *ATOM3AppearanceIcon* are icon-like, and they may include primitive forms such as circles, lines, text and show attribute values of the object associated with the instance through relationship *Appearance*. Instances of *ATOM3AppearanceLink* are similar to the previous one, but are associated with two *ATOM3AppearanceSegment* instances, which represent the incoming and outgoing segments to the link (which is itself drawn in the centre). Finally, the *ATOM3Attribute* class implements a special kind of attribute type, which is an instance of itself. In this way one can have arbitrary meta-modelling layers.

The second level in Figure 1 shows a part of the meta-model defined in Figure 4 (the lower part), but using an *abstract syntax* form (instead of the common graphical appearance of UML class diagrams that we have used in the upper meta-metamodel) where we indicate the elements of the upper meta-level from which they are instances. Only two classes are shown, *ActivationBox* and *Object*, together with the attributes for defining their appearances. In AToM^3, by default, the name of the appearance associated with a class or association begins with *"Graph_"* followed by the name of the class or association (that is, the *name* attribute defined in *ModelElement* is filled automatically). In the case of an *AssociationEnd* instance, it is similar, but followed by an *"S"* or *"T"*, depending if the end is source or target.

Finally, the lowest meta-level shows to the left (using an *abstract syntax* notation) a simple sequence diagram model. To the right, the same model is shown, using a visual representation, taking the graphical appearances designed for *Graph_Object*, *Graph_ActivationBox*, *Graph_LifeLine*, *Graph_LifeLineS* and *Graph_LifeLineT*. Note how the graphical forms are in a one-to-one correspondence with the non-graphical elements (*Object1*, *LL1*, *LLS1*, *LLT1* and *ABox1*). The non-graphical elements can be seen as the *abstract syntax* and the graphical ones as the *concrete syntax*. Nonetheless, as stated in the introduction, the one-to-one relationship is very restrictive. Therefore we propose building two separate meta-models, one for the concrete syntax representation (whose concepts are the graphical elements that the user draws on the screen) and another one for the abstract syntax. Both of them are related using a correspondence graph. The user builds the concrete syntax model, and a (triple, event-driven) graph grammar builds and checks the consistency of the abstract syntax model. These concepts are introduced in the following section.

3   Non-monotonic, Abstract Triple Graph Grammars

Triple Graph Grammars were introduced by Schürr [11] as a means to specify translators of data structures, check consistency, or propagate small changes of one data structure as incremental updates into another one. Triple graph grammar rules model the transformations of three separate graphs: source, target and correspondence graphs. The latter has morphisms from each node into source and target nodes. These concepts can be defined as follows(taken from [11]) [1] :

---

[1] For space limitations, we have skipped all proofs referred to the constructions we introduce.

Definition 1 *(Graph Triple) Let $CONC$, $ABST$ and $LINK$ be three graphs and $gs\colon LINK \to CONC$, $gt\colon LINK \to ABST$ be two morphisms. The resulting graph triple is denoted as: $CONC \xleftarrow{gs} LINK \xrightarrow{gt} ABST$.*

Morphisms gs and gt represent m-to-n relationships between $CONC$ and $ABST$ graphs via $LINK$ in the following way: $x \in CONC$ is related to $y \in ABST \iff \exists z \in LINK \mid x = gs(z)\ and\ y = gt(z)$.

In [11] triple graph grammars were defined following the single pushout [7] (SPO) approach and were restricted to be monotonic (its LHS must be included in its RHS). In this way, only two morphisms were needed from the RHS of the $LINK$ graph to the RHS of the $CONC$ and $ABST$ graphs. Morphisms in LHS are defined thus as a restriction of the morphisms in RHS. Here we use the double pushout approach [7] (DPO) with negative application conditions (NAC) in rules and do not take the restriction of monotonicity. Hence, we have to define two morphisms from both LHS and RHS of the correspondence graph rule to the LHS and RHS of the $CONC$ and $ABST$ graphs.

Definition 2 *(Triple Rule) Let $sp = (SL \xleftarrow{sl} SK \xrightarrow{sr} SR)$, $cp = (CL \xleftarrow{cl} CK \xrightarrow{cr} CR)$ and $tp = (TL \xleftarrow{tl} TK \xrightarrow{tr} TR)$ be three rules. $NAC = \{(NS \xleftarrow{nl} NC \xrightarrow{nr} NT, n)\}$ is a set of tuples where the first component is a graph triple and n is a triple $(n_S\colon SL \to NS, n_C\colon CL \to NC, n_T\colon TL \to NT)$ of injective graph morphisms. Furthermore, let $ls\colon CL \to SL$, $rs\colon CR \to SR$, $lt\colon CL \to TL$ and $rt\colon CR \to TR$ be four graph morphisms, such that they coincide in the elements of CK as follows: $\forall k_1 \in CK, \exists k_2 \in SK, ls(cl(k_1)) = sl(k_2) \land rs(cr(k_1)) = sr(k_2)$ [2] (and analogously for the elements of TK). The resulting triple rule (see Figure 2)is defined as follows: $p = (sp \xleftarrow{ls,rs} cp \xrightarrow{lt,rt} tp, NAC)$.*
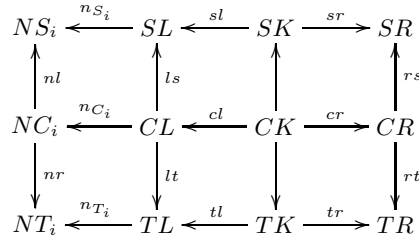

Fig. 2. A triple rule.

Figure 3 shows an example of two triple rules (where the dashed arrows depict morphisms ls, rs, lt and rt) with NACs, where only the additional elements to LHS and their context have been depicted. NACs have the usual meaning, if a match is found in the triple graph (which commutes with the LHS match and n), the rule cannot be applied. The kernel parts SK, CK and TK of the rules are not explicitly shown, but

---

[2] which is equivalent to $\exists cs\colon CK \to SK$ such that $ls \circ cl = sl \circ cs$

their elements have the same numbers in LHS and RHS. This is the notation that we use throughout the paper. For our purposes, we need to extend the previous definition of triple grammars to include attributes. This can be done in the way shown in [11].
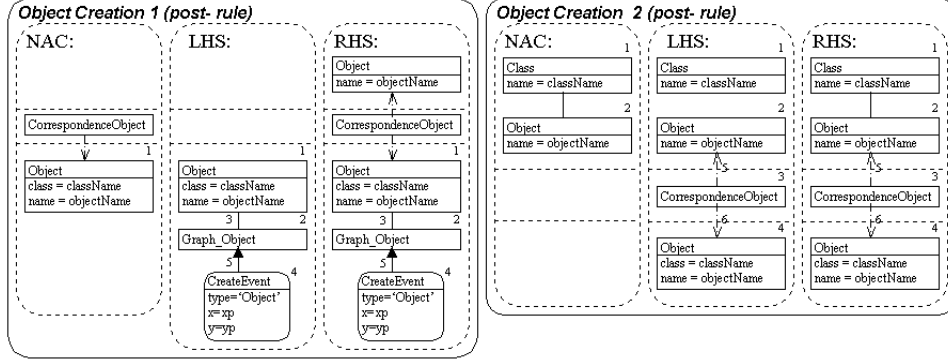

Fig. 3. An Example with two Triple Rules

For the approach to be useful in meta-modelling environments, graphs must be consistent with a meta-model. We model this by defining *typing* morphisms between graphs and *type graphs*. We use the concept of *type graph with inheritance* [3] as defined in [3]:

Definition 3 *(Type Graph with Inheritance, taken from [3]) A type graph with inheritance is a triple (TG, I, A) of graphs TG and I sharing the same set of nodes N, and a set $A \subseteq N$, called abstract nodes. For each node n in I the* inheritance clan *is defined by $clan_I(n) = \{n' \in N \mid \exists\, path\ n' \xrightarrow{*} n\ in\ I\}$ where path of length 0 is included, i.e. $n \in clan_I(n)$.*

For the typing of a graph triple, we have to define meta-models for the $CONC$, $ABST$ and $LINK$ graphs. Additionally, as $LINK$ has morphisms to $CONC$ and $ABST$, we have to include information about the valid morphisms in the meta-model for the $LINK$ graph. Thus, we define a *meta-model triple* in the following way:

Definition 4 *(Meta-model triple) A meta-model triple is a triple of type graphs with inheritance, together with two morphisms (cs and ct) between nodes of one of the type graphs to the other two: $MMT = ((TG^{CONC}, I^{CONC}, A^{CONC}), (TG^{LINK}, I^{LINK}, A^{LINK}), (TG^{ABST}, I^{ABST}, A^{ABST}), cs, ct)$ where $cs\colon TG^{LINK} \to TG^{CONC}$ and $ct\colon TG^{LINK} \to TG^{ABST}$*

Figure 4 shows an example meta-model triple, which in the upper part (abstract syntax) depicts a slight variation of the UML 1.5 standard meta-model proposed by OMG for sequence diagrams. We have collapsed the triples (TG, I, A) into a unique graph,

---

[3] In the following, we indistinctly use the terms "type graph" and "meta-model", although the latter may include additional constraints.

where the I graph is shown with hollow edges (following the usual UML notation) and the elements in A are shown in italics.

The lower meta-model in the figure declares the concrete appearance concepts and their relationships. The elements in this meta-model are in direct relationship with the graphical forms that will be used for graphical representation. As Figure 1 showed, we allow the refinement of relationships, and this is shown with the usual notation for inheritance, but applied to relationships (arrows in the diagram). This is just a notation convenience, because each relationship (arrow) shown in Figure 4 is indeed an instance of class *Association* in the upper meta-model in Figure 1. In this way, the inheritance concept developed in [3] is immediately applicable to refinement of relationships.

The correspondence meta-model formalizes the kind of morphisms that are allowed from nodes of types *CorrespondenceMessage* and *CorrespondenceObject*. As it is defined, the declared morphism types in *cs* and *ct* are not *"inherited"* through I^{LINK} in the correspondence graph meta-model.
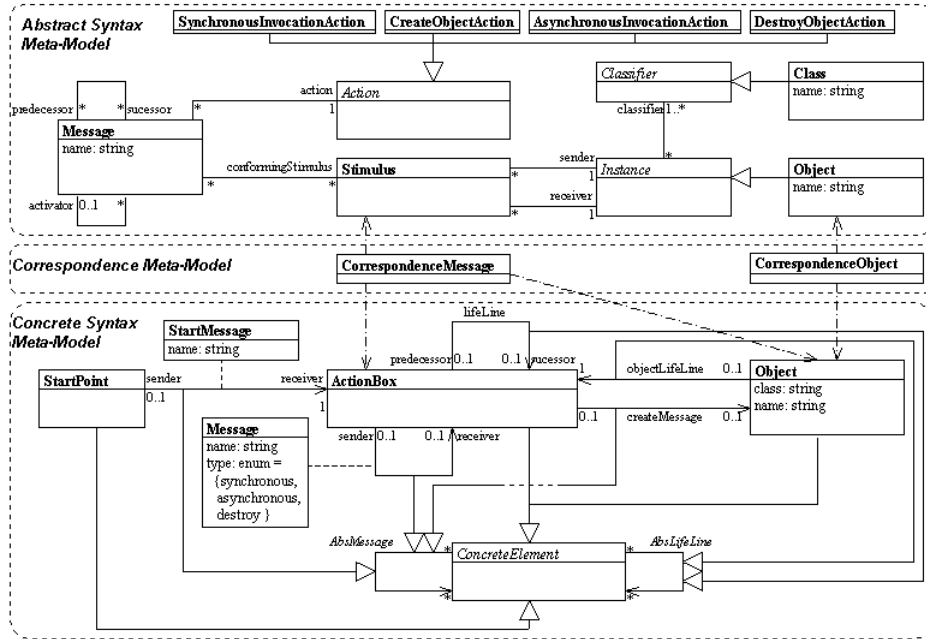

Fig. 4. An Example Meta-model triple

Triple rules must be provided with typing morphisms to the meta-model triple. As in [3] we use the notion of *clan morphism* from graphs to type graphs with inheritance.

Definition 5 *(Clan Morphism, taken from [3]) Given a type graph with inheritance* (TG, I, A) *and graph* G, $type'\colon G \to TG$ *is a* clan-morphism, *if for all* $e \in G_E$ $type'_N \circ s_G(e) \in clan_I(s_{TG} \circ type'_E(e))$ *and similar for* t_G

We can define typed graph triples in a similar way as typed rules were defined in [3], but constraints regarding the morphisms of the correspondence graph should also be given. Additionally, we can define *abstract* triple rules by allowing the appearance of abstract nodes in LHS of each rule. If an abstract node appears in the RHS, then it must also appear in the LHS. An abstract rule is equivalent to a number of concrete rules where each abstract node is replaced by any concrete node in its inheritance clan. For the application of this concept here, first note that an *abstract* triple rule is equivalent to the combination of all its concrete subrules. Additionally, some of these combinations may be not valid, because of invalid morphisms between the resulting concrete rules of the correspondence graph and the source and target graphs.

Definition 6 *(Typed Graph Triple) A graph triple typed by a meta-model triple $MMT = ((TG^{CONC}, I^{CONC}, A^{CONC}), (TG^{LINK}, I^{LINK}, A^{LINK}), (TG^{ABST}, I^{ABST}, A^{ABST}), cs, ct)$ is depicted by $TRIG_{MMT} = (CONC \xleftarrow{gs} LINK \xrightarrow{gt} ABST, type_C, type_L, type_A)$ where the last three components are typing clan morphisms from $CONC$, $LINK$ and $ABST$ to the first three components of MMT in which the following conditions hold: $\forall l \in LINK\ type_C(gs(l)) \in clan_{I^{CONC}}(cs(type_L(l)))$ and $type_A(gt(l)) \in clan_{I^{ABST}}(ct(type_L(l)))$*
If the image of any element of the triple graph belongs to some of the A sets, the typing is called abstract, otherwise it is called concrete.

Definition 7 *(Abstract Triple Rule) A triple rule typed by a meta-model triple MMT (defined as before) is depicted by $TRIP_{MMT} = (sp \xleftarrow{ls,rs} cp \xrightarrow{lt,rt} tp, NAC, type_{sp}, type_{cp}, type_{tp})$ where $type_{sp}$ is a triple of clan morphisms ($type_{sp}^L$, $type_{sp}^K$ and $type_{sp}^R$) from SL, SK and SR ($sp = (SL \xleftarrow{sl} SK \xrightarrow{sr} SR)$) to TG^s (and similar for $type_{cp}$ and $type_{tp}$). Additionally, NACs are also typed as follows: $NAC = \{(NS \xleftarrow{nl} NC \xrightarrow{nr} NT, n, type^N)\}$ is a set of tuples where the first two components are defined as in definition 2 and $type^N$ is a triple of clan morphisms ($type_S^N$, $type_C^N$, $type_T^N$) from the graph triple to TG^s, TG^c and TG^t, which forms a typed graph triple with the first component (see definition 6).*
The following conditions hold for sp:

- *$type_{sp}^L \circ sl = type_{sp}^K = type_{sp}^R \circ sr$ (typing of preserved elements do not change).*
- *$type_{sp,N}^R(R'_{sp,N}) \cap A^s = \emptyset$, where $R'_{sp,N} := SR_N - sr_N(SK_N)$ (new nodes in RHS are not abstract)*
- *$type_S^N \circ n_S \leq type_{sp}^L$ for all $(N, n, type^N) \in NAC$ (where $\leq$ is the type refinement relationship [3]) (typing for NACs is finer than the corresponding elements in LHS)*

And analogously for cp and tp. As in previous definition, $\forall n \in CL, type_{sp}^L(ls(n)) \in clan_{I^{CONC}}(cs(type_{cp}^L(n)))$ and $type_{tp}^L(lt(n)) \in clan_{I^{ABST}}(ct(type_{cp}^L(n)))$ (and analogously for CK and CR)

Once we have defined the basic concepts regarding graph rules, next section presents event-driven grammars, which we use in combination with abstract triple rules in order to build the abstract syntax model associated with the concrete syntax. They are also useful for consistency checking, as we will see in section 5.

4 Event-Driven Grammars

In this section, we present *event-driven grammars*, as a means to formalize some of the user actions and their consequences when using a visual modelling tool. We have defined event-driven grammars to model the effects of editor operations in AToM [3] [6], although other tools could also be modelled. The actions a user can perform in AToM [3] are *creating*, *editing* and *deleting* an entity or a connection, and *connecting* and *disconnecting* two entities. All these events occur at the concrete syntax level.

The main idea of *event-driven* grammars is to make explicit these events in the models. Note how this is very different from the *syntax directed* approach, where graph grammar rules are defined for VL generation. In these environments the user chooses the rule to be applied. In our approach, the VLs are generated by means of meta-modelling, and the user builds the model as in regular environments generated by meta-modelling (*free-hand editing*). The events that the user generates may trigger the execution of some rules. In our approach, rules are triple rules and are used to build the abstract syntax model and to perform consistency checkings.

We have defined a set of rules (called *event-generator rules*, depicted as *evt* in Figure 5) that models the generation of events by the user. Another set of rules (called *action rules*, depicted as *sys-act* in Figure 5) models the actual action triggered by the event (creating, deleting entities, etc.), and finally, an additional set of rules (called *consume rules*, depicted as *del* in Figure 5) models the consumption of the events once the action has been performed. The VL designer can define his own rules to be executed after an event and before the execution of the *action rules* (depicted as *pre* in Figure 5), or after the *action rules* and before the *consume rules* (depicted as *post* in Figure 5)). These rules model pre- and post- actions respectively. In the pre-actions, rules can delete the produced events, if certain conditions are met. This is a means to specify pre-conditions for the event to take place. Additionally, in the post-actions, rules can delete the event and undo its actions, which is similar to a post-condition. The working scheme of an event-driven grammar is shown in Figure 5. All the sets of rules, (except the ones in *evt*, which just produce a user event) are executed as long as possible.

$$M_i \xrightarrow{evt} M_{evt} \xrightarrow{pre*} M_{evt-pre} \xrightarrow{sys-act*} M_{act} \xrightarrow{post*} M_{act-post} \xrightarrow{del*} M_f$$

Fig. 5. Application of an event driven grammar with user-defined rules.

In the example presented in this paper, models (M_i, M_{evt}, $M_{evt-pre}$, M_{act}, $M_{act-post}$ and M_f in Figure 5) are indeed typed graph triples. In this way, the set of rules evt, $sys - act$ and del are applied to the $CONC$ graph, which represents the concrete syntax. In the example, rules in *pre* and *post* are abstract triple rules, used to propagate the changes due to the user-generated events to the abstract syntax model ($ABST$ graph).

Figure 6 shows the AToM[3] base classes for the concrete syntax. As stated before, all concrete syntax symbols inherit either from *Entity* (if it is an icon-like entity) or from *Link* (if it is an arrow-like entity). Both *Entity* and *Link* inherit from *VisualObject*,

which has information about the object's location (*x* and *y*) and about if it is being dragged (*selected*). *Links* are connected to *Entities* via *Segments*; these can go either from *Entities* to *Links* (*e2l*) or the other way around (*l2e*).
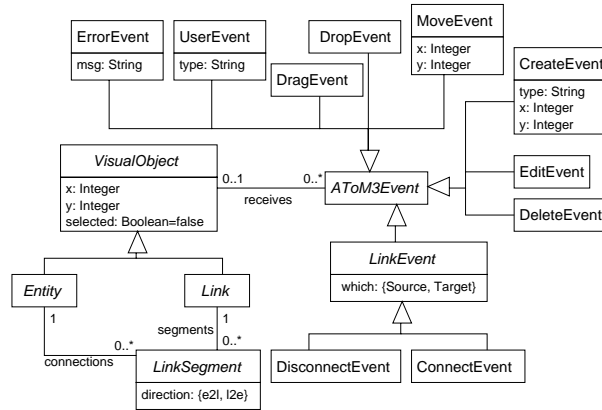

Fig. 6. AToM³ base classes for concrete syntax objects and user events.

Some of the classes in Figure 6 model the events that can be generated by the user. All the events can be associated to a *VisualObject*. Some events have additional information, such as *CreateEvent*, which contains the type of the *VisualObject* to be created, and its position. The *MoveEvent* contains the position where the object has been moved. When connecting two *Entities*, two *ConnectEvent* objects are generated, one associated to the source and other one associated to the target. *ErrorEvent* signals an error associated with a certain object, AToM³ presents the text of the error and highlights the associated object. Finally, the *UserEvent* class can be used to define new events.
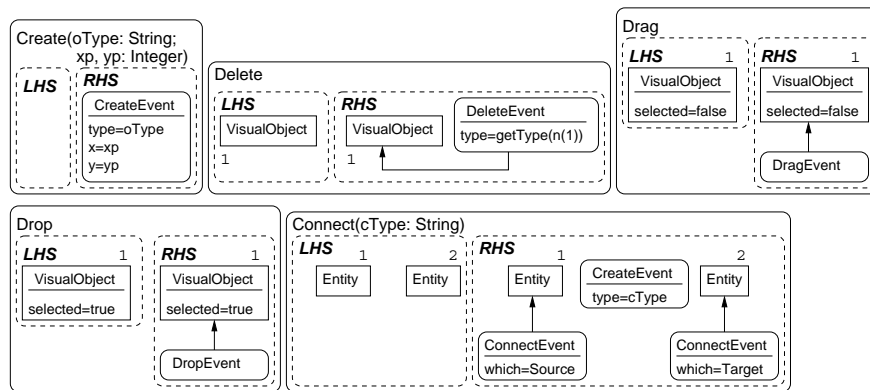

Fig. 7. Some of the *event-generator* rules.

Figure 7 shows some of the *event-generator* rules (depicted as *evt* in Figure 5), which model the generation of events by the user. The *Create* rule is triggered when the user clicks on the button to create a certain entity, and then on the canvas. The type of the object to be created is given by the button that the user clicks, and the *x* and *y* coordinates by the position of the cursor in the canvas. In AToM^3, a button is created for each non-abstract class in the meta-model. The *Delete* rule is triggered when the user deletes an object. The type of the object to be deleted is obtained by calling the *getType* function on node number one. This is a function which is available in Python (the implementation language of AToM^3) and returns the actual type of an object. Finally, the *Connect* rule is invoked when the user connects two *Entities*. In AToM^3 this is performed by clicking in the *connect* button and then on the source and the target entities. AToM^3 infers (with the meta-model information) the type of the subclass of *Link* that must be created in between. If several choices exist, then the user selects one of them. The type is then passed as a parameter of the rule, and the corresponding creation event is generated.
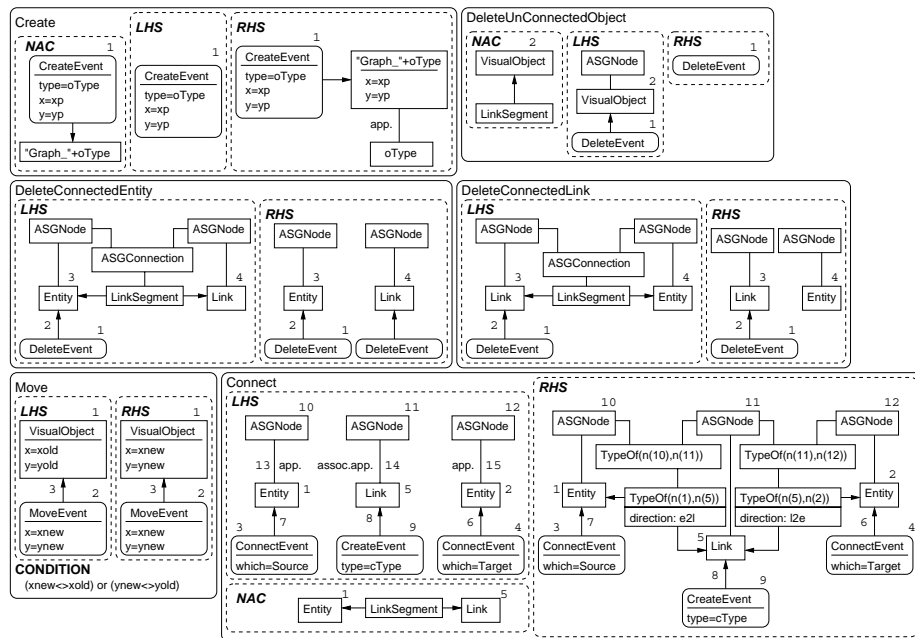

Fig. 8. Some of the *action* rules.

Figure 8 shows some of the rules that model the actual execution of the events (depicted as *sys-act* in Figure 5). The first rule models the actual creation of an instance (subclass of *ASGNode*, see Figure 1), together with its associated visual representation (whose type name is the same as the non-visual instance, but starting by *"Graph_"* and is a subclass of *Link*). The three following rules model the execution of a *delete* event. In the first case (*DeleteUnConnectedObject* rule), the object has no connections. In the

second case (*DeleteConnectedEntity* rule), the icon-like object has connections, so a *delete* event is sent to the connected link, and the segment is erased. The third case (*DeleteConnectedLink* rule) models the deletion of a link (the "centre" of an arrow-like graphical form), which also deletes the associated segment. Please note that all the rules are executed as long as possible (see Figure 5). The *Move* rule simply modifies the position attributes of the object. Finally, the *Connect* rule models the connection of a link to two entities. Note, that rule *Connect* in Figure 7 generates a *CreateEvent* for the *link*, so rule *Create* in Figure 8 is executed first. The rule creates the *link* with the correct type. Next, rule *Connect* in Figure 8 can be applied, as classes *Entity* and *Link* are the base classes for all graphical objects. Note how the appropriate types for the segments in between links and entities are obtained (from the AToM^3 API) through function *TypeOf* which searches the information in the meta-model. Finally, a last set of rules (not shown in the paper) models the deletion of the events.

5   Example: Sequence Diagrams

As an example of the techniques explained before, we have built an environment to define UML sequence diagrams. By means of meta-modelling we define the abstract and concrete syntax of this kind of diagrams, as well as the correspondence relation between their elements (see meta-model triple in Figure 4). Starting from this triple meta-model, AToM^3 generates a tool where the user can build models according to that syntax. The user creates the diagrams at the concrete syntax level, therefore some automatic mechanism to generate the abstract syntax of the diagrams and support its mutual coherence has to be provided. With this aim we have built a set of event-driven rules triggered by user actions. Additionally, another set of triple rules check the consistency between the sequence diagram and existing class diagrams. Both set of rules are presented in the following subsections.

5.1   Abstract and Concrete Syntax of Sequence Diagrams

These rules manage the creation, edition and deletion of *Objects*, the creation, edition and deletion of *Messages*, and the creation and deletion of object *Life Lines*. The graphical actions that do not change the diagram abstract syntax (like creating an *Activation Box*) do not need the definition of extra event rules apart from the ones provided by AToM^3 (see Figures 7 and 8).

Rules for the creation, edition and deletion of *Objects* are the simplest of the set. These rules create, edit and delete *Objects* at the abstract syntax level (once the user generates the corresponding event at the concrete level). *Objects* at the abstract syntax are related to the concrete syntax *Objects* (which received the user event) through an element in the correspondence graph. Rules for creating objects (both *post-* actions, see Figure 5) are shown in figure 3. The rule on the left creates the object at the abstract syntax level, while the rule on the right connects (at the abstract syntax level) the object with its corresponding class. If the rule on the right cannot be applied, it means that the object class has not been created in any class diagram. This inconsistency is tolerated at this moment (we do not want to put many constraints in the way the user builds the

different diagrams), but we have created a grammar to check and signal inconsistencies, including this one. The grammar is explained in the next subsection and can be executed at any moment in the modelling phase. For the deletion of an object (rules not shown in the paper), we ensure that it has no incoming or outgoing connection. This is done by a *pre-* condition rule (not shown in this paper) that erases the *delete* event on an object and presents a message if it has some connection.

The creation of a message is equivalent to connecting two elements belonging to the concrete syntax (*ConcreteElement*, see Figure 4) by means of a relationship of type *AbsMessage*. Obviously users cannot instantiate neither abstract entities nor abstract relationships, but only concrete ones. Therefore, at the user level the action to create messages includes three concrete cases: the connection of two *Activation Boxes* by means of a *Message* relation, the connection from an *Activation Box* to an *Object* by means of a *createMessage* relationship, and the connection from a *Start Point* to an *Activation Box* by means of a *startMessage* relation. The event rules for managing these three concrete cases are very similar except for the entities and relationships participating in the action. That is, we should have a first rule to create a *Message* relationship if its source and target are activation boxes; a second identical rule except for the relationship type (*createMessage*) and the target of the relationship (*Object*); and a third similar rule except for the relationship type (*StartMessage*) and source (*Start Point*). Since the three rules have the same structure, we use an abstract rule to reduce the grammar size. In Figure 9 we show the abstract rule compressing the first and third concrete rules mentioned above. We have used abstraction in many other rules, which highly reduces the total amount of rules. The rule in Figure 9 generates the abstract syntax of a new message created by the user, establishing a morphism between the concrete syntax of the new message (graphical appearance) and its respective abstract syntax. In this particular case the message concrete syntax is related to more than one abstract syntax entity: three abstract syntax entities (one *Message*, one *Stimulus* and one *Action*)) are graphically represented using a single symbol on the concrete syntax. On the other hand, the same event rule has to process the relationship between the newly created message and the rest of the model. In this way the successor, predecessor and activator messages of the created one have to be computed, as well as the objects sending and receiving the message. Additionally, we have to check if the new message activates in its turn another block of messages. We have broken down the creation event in a set of 6 user-defined events, each performing one step in the process. Thus the number of rules is reduced and the processing is easier.

Other rules (not shown in the paper) calculate the *predecessor* of a message. This is the previous one in the same processing block (the activation boxes corresponding with a method execution), or none if the message is the first one in the block. A total of 16 rules have been defined to manage the creation and edition of *objects* and *messages*. Some other rules, similar to the previous ones, manage the creation and deletion of *Life Lines*. The processing of the event (creation or deletion) triggers the execution of other user-defined events, simpler to process. Most of these events are the same as the ones generated by rule in Figure 9, therefore reutilization of rules has been possible. Due to space limitation, we do not show all the rules, which are 39 in total.
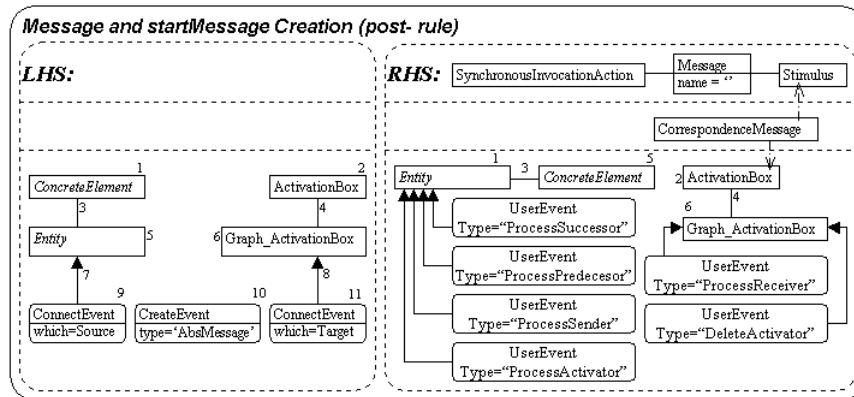
Fig. 9. Abstract rule for Creating *Messages* and *createMessages*.

5.2 Consistency Checking

Triple rules can be used not only to maintain coherence between concrete and abstract syntax, but also to check consistency between different types of diagrams. The present work is part of a more general project with the aim to formalize the dynamic semantics of UML [8] by means of transformations into semantic domains (up to now Petri nets). Before translation, consistency checkings should be performed between the defined diagram (in this case a sequence diagram) and existing ones, such as class diagrams. Note how, while the user builds a sequence diagram, the previous rules add abstract syntax elements to a unique abstract syntax model. In this way, one has a unique abstract syntax model and possibly many concrete syntax models, one for each defined diagram (of any kind).

Using simple triple rules, we can perform consistency checkings between the sequence diagram and an existing abstract syntax model, generated by previously defined diagrams. For example, we may want to check that the class of the objects used in a sequence diagram has been defined in some of the existing class diagrams; if an object invokes a method of another object, the method should have been defined in its class, and there should be a navigable relationship between both object classes (see Figure 10), and that the invoked method is visible from the calling class.

We define *consistency triple rules* in such a way that their LHSs are conditions that are sought in the defined diagram (sequence diagrams in our case), possibly in the concrete and abstract parts. NACs are typically conditions to be sought in the existing abstract model with which we want to check consistency. If the rule is applied the rule's RHS sends an event of type *ErrorEvent* to some of the objects matched by the LHS.

6 Related Work

At a first glance, the present work may resemble the *syntax directed approach* for the definition of a VL. In this approach one defines a rule for each possible editing action,
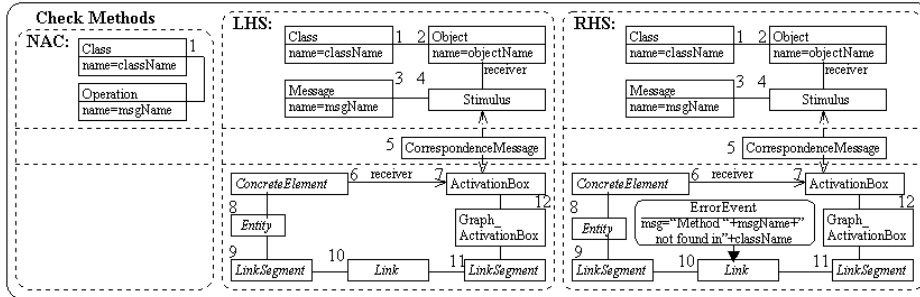
Fig. 10. One of the Rules for Consistency Checking.

and the user builds the model by selecting the rules to be applied. Our approach is quite different, in the sense that we use a meta-model for the definition of the VL. The meta-model (which may include some constraints) provides all the information needed for the generation of the VL. The user builds the model by interacting with the user interface. In our approach we explicitly represent these events in the rules. Rules are triggered by the events, but the user may not be aware of this fact. In the examples, we have shown the combination of event-driven grammars with triple grammars to build the abstract syntax model and to perform consistency checks.

In the approach of [4], a restricted form of Statecharts was defined using a pure graph grammar approach (no meta-models). For this purpose, they used a *low level* (LLG, concrete syntax) and a *high level* (HLG, abstract syntax) representation. To verify the correctness, the LLG has to be transformed into an HLG (using a regular graph grammar), and a parsing grammar has to be defined for the latter. Other parsing approach based on constraint multiset grammars is the one of CIDER [9].

Other approaches for the definition of the VLs of the different UML diagrams, usually concentrate either on the concrete or the abstract syntax, but not on both. For example, in [5], graph transformation units are used to translate from sequence diagrams into collaboration diagrams. Note how, both kind of diagrams share the same abstract syntax, so in our case, a translation is not necessary, but we have to define triple rules to build the abstract syntax from the concrete one.

7   Conclusions

In this paper we have presented *event-driven grammars* in which user interface events are made explicit, and system actions in response to these events are modelled as graph grammar rules. Their combination with abstract triple rules and meta-modelling is an expressive means to describe the relationships between concrete and abstract syntax models (formally defined through meta-models). Rules can model pre- and post- conditions and actions for events to take place. Furthermore, we can use the information in the meta-models to define *abstract* rules, which are equivalent to a number of concrete ones, where nodes are replaced by each element in its inheritance clan. In this work, we

have extended (in a straightforward way) the original work in [3] to allow refinement of relationships.

The applicability of these concepts has been shown by an example, in which we have defined a meta-model triple for the abstract and concrete syntax of sequence diagrams (according to the UML 1.5 specification). Additionally, we have presented some rules to check the consistency of sequence diagrams models with an existing abstract syntax model, generated by the previous definition of other diagrams.

Regarding future work, we want to derive validation techniques for triple, event-driven grammars. We also plan to use triple graph grammars to describe heuristics for the creation of UML diagrams. For example, if the user creates an object in a sequence diagram which belongs to a non-existing class, one option is to raise a consistency warning. Other possibility is to automatically derive the concrete syntax of a class diagram with the information of the abstract syntax (classes, methods, etc.) generated by the sequence diagram.

References

1. Atkinson, C., Kühne, T. 2002. *Rearchitecting the UML infrastructure*. ACM Transactions on Modeling and Computer Simulation, Vol 12(4), pp.: 290-321.
2. Bardohl, R. 2002. *A Visual Environment for Visual Languages.* Science of Computer Programming 44, pp.: 181-203. See also the GENGED home page: `http://tfs.cs.tu-berlin.de/~genged/`.
3. Bardohl, R., Ehrig, H., de Lara J., and Taentzer, G. 2004. *Integrating Meta Modelling with Graph Transformation for Efficient Visual Language Definition and Model Manipulation*. In proceedings of ETAPS/FASE'04, LNCS 2984, pp.: 214-228. Springer.
4. Bottoni, P., Taentzer, G., Schürr, A. 2000. *Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation* Proc. of VL'2000, pp.: 59–60.
5. Cordes, B., Hölscher, Kreowski, H-J. 2003. *UML Interaction Diagrams: Correct Translation of Sequence Diagrams into Collaboration Diagrams*. Proc. of AGTIVE'03, pp.: 273-288.
6. de Lara, J., Vangheluwe, H. 2002. *AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling.* In ETAPS/FASE'02, LNCS 2306, pp.: 174 - 188. Springer-Verlag. See also the AToM^3 home page at: `http://atom3.cs.mcgill.ca`
7. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. 1999. *Handbook of Graph Grammars and Computing by Graph Transformation.* (1). World Scientific.
8. Guerra, E., de Lara, J. 2003. *A Framework for the Verification of UML Models. Examples using Petri Nets.* Jornadas de Ingeniera del Software y Bases de Datos, JISBD. Alicante. Spain. pp.: 325-334.
9. Jansen, A.R, Marriott, K. and Meyer, B. 2003. *CIDER: A Component-Based Toolkit for Creating Smart Diagram Environments.* Proc. of the 9th Conference on Distributed and Multimedia Systems. pp.: 353-359. Knowledge Systems Institute.
10. Odell, J. 1994. *Power types*. Journal of Object Oriented Programming (May).
11. Schürr, A. 1994. *Specification of Graph Translators with Triple Graph Grammars.* In LNCS 903, pp.: 151-163. Springer.
12. UML specification at the OMG's home page: `http://www.omg.org/UML`.