Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Event-Driven Grammars: Relating Abstract and Concrete Levels of Visual Languages

Esther Guerra[1], **Juan de Lara**[2]⋆

[1] Dept. Informática
Universidad Carlos III
Madrid, Spain
e-mail: eguerra@inf.uc3m.es

[2] Escuela Politécnica Superior, Ing. Informática
Universidad Autónoma
Madrid, Spain
e-mail: jdelara@uam.es

Received: date / Revised version: date

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 their turn may trigger other user-interface events. Their combination with triple graph transformation systems allows constructing and checking the consistency of the abstract syntax graph while the user is building the concrete syntax model, as well as managing the layout of the concrete syntax representation.

As an example of these concepts, we show the definition of a modelling environment for UML sequence diagrams. A discussion is also presented of methodological aspects for the generation of environments for visual languages with multiple views, its connection with triple graph grammars, the formalization of the latter in the double pushout approach and its extension with an inheritance concept.

Key words Graph Grammars – Triple Graph Transformation – 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* [10]. The former approach requires the construction of 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 and 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. 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 [8] with inheritance [3], multiplicities and other – possibly textual – constraints. One of the most prominent examples of the meta-modelling approach is the definition of the UML language by the OMG [31]. The meta-modelling environment has to check that the model built by the user is conformant to the meta-model. This can be done by finding a typing morphism between model and meta-model, and by checking the defined constraints on the model. Most of the times, the concrete syntax is given by assigning graphical appearances to both classes and relationships in the meta-model [10]. For example, in the AToM^3 tool [10], 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. Therefore, it is difficult to provide the meta-model with a concrete syntax that is structurally different from the abstract syntax. Moreover, for some applications, one is interested in having several concrete syntax representations for a single meta-model. For example, in the UML1.5 [31], sequence and collaboration diagrams are two different visualizations of the same abstract syntax elements.

In this paper we present a novel approach for VLs definition that combines the meta-modelling and the graph

Send offprint requests to:
⋆ *This is a revised and extended version of a paper presented at the ICGT'04 conference, see [21]*

grammar approaches. To overcome the restriction of a one-to-one mapping between abstract and concrete syntax elements, we define separate meta-models for both kinds of syntax. In a general case, both kinds of models can be very different. For example, in the definition of UML 1.5 class diagrams [31], the meta-model defines abstract syntax concepts *Association* and *AssociationEnd*[1], which are graphically represented together in a single concrete syntax concept (a line). In general, one can have abstract syntax concepts that are not represented at all, represented with a number of concrete syntax elements, and finally, concrete syntax elements without an abstract syntax representation. 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.

In our approach, the concrete syntax part works in the same way as in the pure meta-modelling approach, but we define triple graph transformation rules [27,23] to automatically build the abstract syntax model from the concrete one, 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 the events that the user generates when working with the editor. These *event-driven grammars* are a very useful specification technique for user interaction and dialog with the generated modelling environment [4].

Additionally, we take advantage of the inheritance structure in the meta-model, and allow the definition of *inheritance-extended triple rules* [3]. Some of the nodes in these inheritance-extended rules may be instances of classes with subtypes (i.e. classes from which a number of children classes inherit). These rules are equivalent to a number of *concrete rules* obtained from the valid substitutions of such nodes by instances of the sub-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 based on AToM^3. We define the concrete and abstract syntax of sequence diagrams, a grammar to maintain the consistency of both syntaxes, with additional rules for concrete syntax layout, and consistency rules to check the sequence diagram against other existing diagrams.

The main contribution of the paper is proposing a formal method (based on graph transformation) to overcome the limitations of current approaches to handle concrete and abstract syntaxes, especially when they are very different. Our approach has the advantage of being graphical and formal, so there is no need to code in low-level languages. In contrast, in other approaches [33], the VL designer needs to know the implementation language and the API (Application Program Interface) of the tool. Moreover, in our approach, the behaviour of the tool itself is modelled by graph transformation rules (the event-driven grammars), which promotes flexibility and makes tool evolution easier. Thus, the user has the possibility to change the tool behaviour, for example to support the "action-object" paradigm of interaction (first selecting an operation and then the object to which the operation is performed), or the "object-action". Moreover, the formal definition of graph transformation makes tool behaviour subject to analysis. This can be useful to detect for example if a given action (modelled by rules) may yield different results or is terminating. Finally, another important contribution of this work is the formalization and extension of triple graph grammars [27] to triple graph transformation systems with node and edge inheritance [23] as well as application conditions using the double pushout approach (DPO) [13] to graph transformation. This formal basis is essential, as triple graph grammars are becoming increasingly popular for expressing model transformation [28].

The rest of the paper is organized as follows. In section 2 we introduce meta-modelling in the context of the AToM^3 tool. Section 3 presents triple graph grammars with our extensions to include application conditions and typing with respect to a type graph with node and edge inheritance. This section introduces the concepts in an intuitive way, by means of examples. The rigorous definitions of the theory are left to Appendix A. Section 4 introduces the main concepts of event-driven grammars, while section 5 presents an example to define the abstract and concrete syntax of sequence diagrams (according to the UML 1.5 specification). Here we also present layout and some consistency rules that check that the elements in the diagram are consistent with already existent elements, defined in other diagrams. Section 6 discusses the implementation of the presented concepts in the AToM^3 tool. Section 7 compares the present work with related research. Finally, section 8 ends with the conclusions and future work. Two additional appendices present the main concepts of the theory we have developed for triple graph transformation and event-driven grammars. For a complete presentation of the theory, the reader is referred to [23].

2 Meta-modelling in AToM^3

AToM^3 [10] is a meta-modelling tool that was developed in collaboration with 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* approach is followed where each element of the

[1] In the UML2.0 version *AssociationEnd* is no longer present, and the *Property* metaclass is used instead (see the UML2.0 superstructure specification [31]).

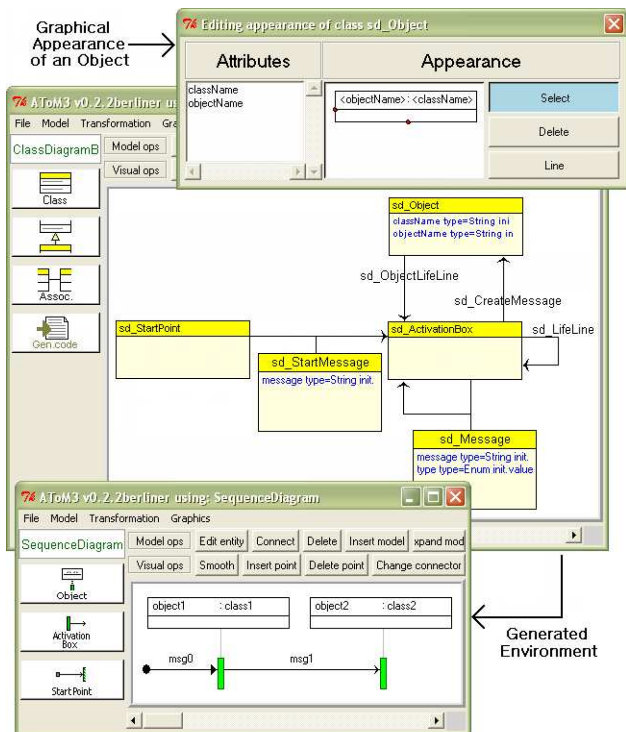

Fig. 1 The AToM^3 Tool.

meta-modelling level n is an instance of exactly one element of the level $n+1$[2] [1]. Starting from the meta-model of a VL, and assigning visualization information to each element in the meta-model, the tool is able to generate a customized modelling environment for the VL.

As an example, the upper part of Fig. 1 shows the AToM^3 tool containing the meta-model of a subset of sequence diagrams. We have included the main visualization concepts of a sequence diagram (messages, objects, activation boxes, life lines) in the meta-model. We have also added attributes to the concepts, including their graphical representation. For example, the graphical representation of an object is shown in the upper-right window in the figure. The modelling environment automatically generated from this definition is shown below. Note that should we have used the meta-model proposed in the UML 1.5 standard specification, it would have been difficult to generate such environment using meta-modelling. The reason is that the concepts in the standard UML specification are quite different from the real visualization. For example, there is no notion of activation boxes or life lines in this standard meta-model. In addition, in the standard meta-model, messages are related through successor and activator relations, which are not explicitly set by the user when using a tool with the concrete-syntax elements. Similar problems arise in the UML 2.0 [31] specification for sequence, time and other diagrams.

[2] This is not exactly true, as for implementation we allow instances to inherit from some base classes (see Fig. 2) that do not have a corresponding concept at the upper meta-level.

Next, we explain the AToM^3 design structure concerning meta-modelling, as it will be used in the following sections. Fig. 2 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. It can be noted that *Associations* can also be refined, and that the types of attributes are specific AToM^3 types. Instances at a lower meta-level of some of the concepts in this meta-metamodel 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. Their instances at a lower meta-level inherit from abstract classes *Entity*, *LinkSegment* and *Link*. The user can define the visual appearance of these instances with a graphical editor (such as the one shown in the upper-right corner in Fig. 1). Instances of *ATOM3AppearanceIcon* are icon-like, and they may include primitive forms such as circles, lines and 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 center). Finally, the *ATOM3Attribute* class implements a special kind of attribute type, which is used to define attribute types (relation "type"). At a lower meta-level, the instance of an *ATOM3Attribute* is of the type pointed to by relation "type" at the upper meta-level. As "type" may point to an *ATOM3Attribute*, it is possible to have an arbitrary number of meta-modelling layers.

The second level in Fig. 2 shows a part of the meta-model presented in Fig. 1. In this second level we have used an *abstract syntax* notation, instead of the common graphical appearance of UML class diagrams that we have used in the upper meta-metamodel. In this level, nodes are labelled with 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. 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_ObjectLifeLine*, *Graph_ObjectLifeLineS* and *Graph_ObjectLifeLineT*. The graphical forms are in a one-to-one

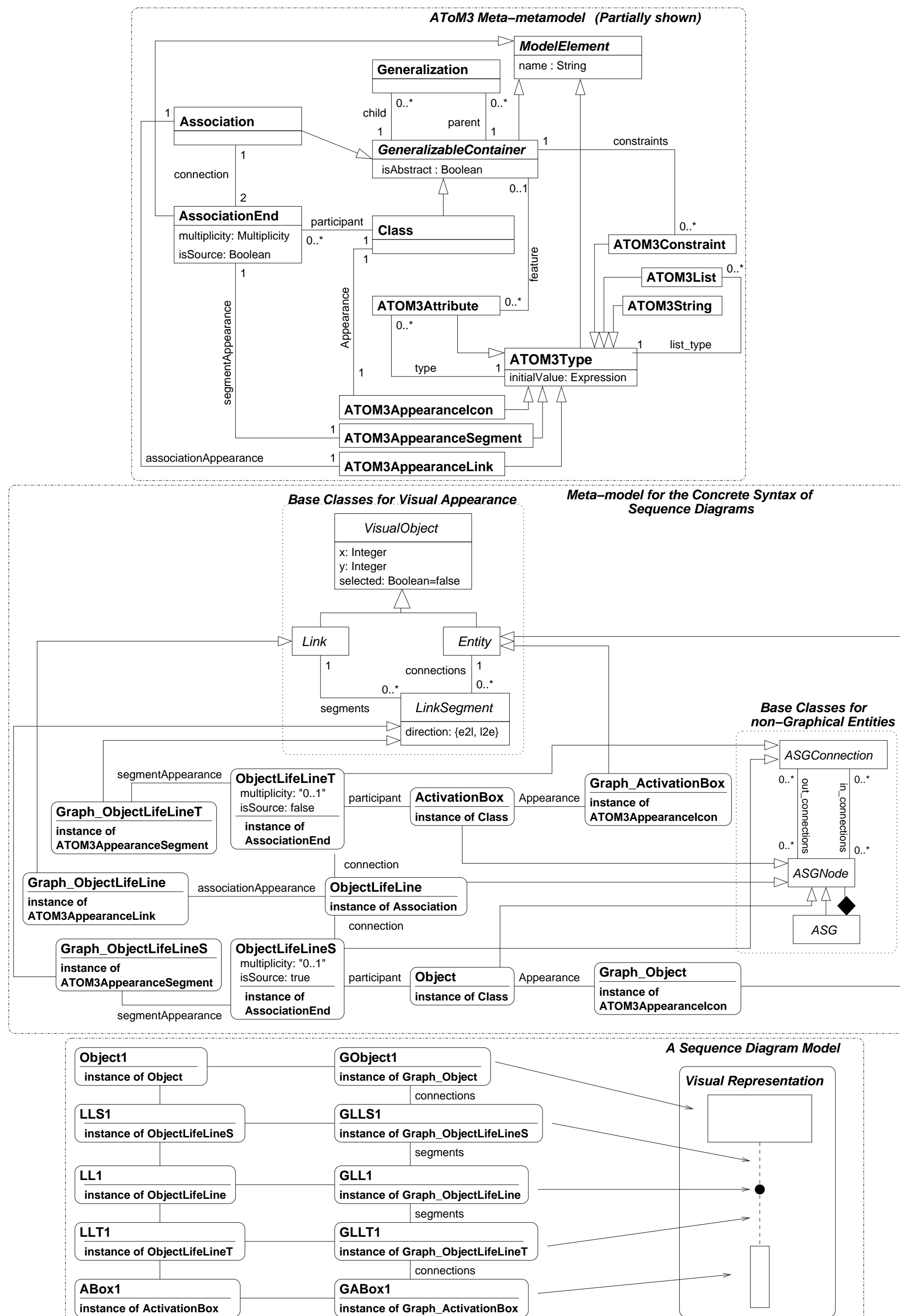

Fig. 2 Meta-modelling Levels in AToM^3.

correspondence with the non-graphical elements (*Object1*, *LL1*, *LLS1*, *LLT1* and *ABox1*). The non-graphical elements can be seen as the *abstract syntax* model and the graphical ones as the *concrete syntax*. Nonetheless, as stated before, 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 one. Both of them are related using a correspondence graph. For example, in the case of UML sequence diagrams, the abstract syntax meta-model contains the standard UML 1.5 definition. In the concrete meta-model we place visualization concepts such as activation boxes or life lines (as shown in the meta-model of Fig. 1). The user builds the concrete syntax model, and a (triple, event-driven) graph grammar builds and checks the consistency of the abstract syntax model. The main

concepts of graph grammars and triple graph transformation systems are introduced in the following section.

3 Attributed Typed Triple Graph Grammars with Node and Edge Inheritance

In this section, we show in an intuitive way the main extensions we have made to triple graph grammars in order to be able to relate arbitrary concrete and abstract syntax. The main concepts are presented in a theoretical way in Appendix A. For a full description of the formalization, the interested reader can consult [23].

Triple Graph Grammars (TGGs) were introduced by Schürr [27] as a means to specify translators of data structures, check consistency, or propagate small changes of a data structure as incremental updates into another one. TGGs manipulate triple graphs; therefore we introduce this structure in subsection 3.1. Then, in subsection 3.2, we present the main ideas of triple graph transformation with node and edge inheritance.

3.1 Attributed Typed Triple Graphs

TGG rules model the transformations of triple graphs made of three separate graphs: source, target and correspondence. As originally defined, nodes in the correspondence graph had morphisms (mappings) to nodes in the source and target graphs. We have extended the notion of triple graph by allowing attributes on nodes and edges (as for example in UML both classes and associations have attributes). Moreover, the relation between source and target graphs is more flexible, as we allow the morphisms from nodes in the correspondence graph to be undefined, or to lead either to a node or an edge. Finally, we also provide triple graphs with a typing by a triple type graph (similar to a triple meta-model), which may contain inheritance relations between nodes or edges.

Fig. 3 shows an example of an attributed typed triple graph (short ATT-graph). The three graphs making the triple graph are separated by dotted lines. The lower graph is called source or concrete, the upper one is called target or abstract, while the graph in the middle is called correspondence and is used to relate elements in the other two graphs. Nodes in the correspondence graph are provided with two morphisms (called correspondence functions) which can reach either a node or an edge of the source and target graphs, or be undefined. For a precise definition of ATT-graph, see definition 7 in Appendix A.

Using a UML-like notation, the lower graph in the figure depicts a small sequence diagram that uses the visualization concepts shown in the meta-model of Fig. 1. In particular, it shows two objects ("object1" and "object2") having an activation box each. The first activation box receives the start message "msg0" and sends message "msg1" to the second one. Note that messages are represented as links with attributes (i.e. attributed edges). The upper graph uses concepts of the UML 1.5 meta-model (*Stimulus, Object, Class*, etc). The correspondence graph in between relates the elements in both graphs. More in detail, nodes with type *CorrespondenceObject* in the correspondence graph relate nodes of type *Object* in the other two graphs, while *CorrespondenceMessage* nodes relate messages of any kind (edges) with *Stimulus* objects (nodes).

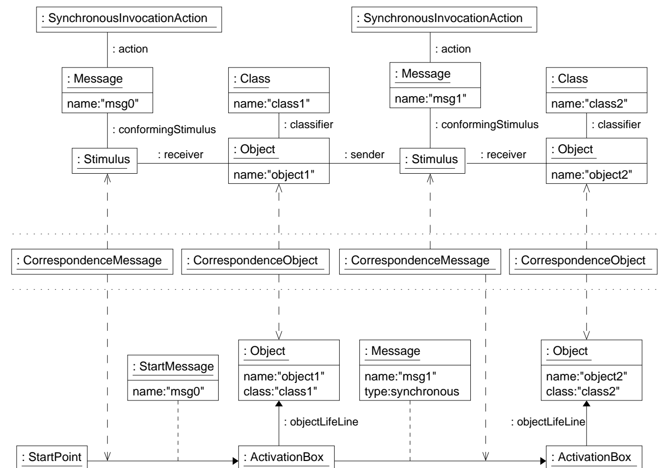

Fig. 3 Attributed Typed Triple Graph Example.

Being able to relate links with both nodes and links through correspondence graph nodes is crucial in our approach. For example, suppose we have two attributed edges with the same source and target nodes in the concrete graph, that we want to relate with other attributed edges in the abstract graph. Relating only the source and target nodes is not enough, as then we do not know which edge in the concrete graph is related to which one in the abstract graph. Therefore it is necessary to be able to directly map edges. Moreover, sometimes it is necessary to relate edges in one graph to nodes in the other (and therefore a regular graph morphism from the correspondence graph to the other two graphs is not enough either). For example, in model transformation, if we translate state automata into Petri nets, we may model automaton transitions as edges and Petri net transitions as nodes. For this transformation, we map states into places and state automaton transitions into Petri net transitions. This kind of *heterogeneous mapping* has been necessary in many other cases, such as in [24], where we transformed the structure of a web system into a coloured Petri net, where hyperlinks (edges) were mapped into Petri net transitions (nodes). Fig. 3 is also an example of heterogeneous mapping.

On the other hand, as the user interacts with the concrete graph, he may delete elements which are already related to elements in the abstract graph. When such operation is performed, the mapping from the correspondence node to the concrete graph node becomes

undefined. Keeping the correspondence node with just one mapping is useful as we may later want to delete the element in the abstract graph, and probably some others related to it. Note that this feature facilitates designing incremental transformations. Moreover, being able to know that a mapping is undefined is a very useful negative test in TGG rules.

The ATT-graph in Fig. 3 is typed over the attributed type triple graph with inheritance (or meta-model triple) shown in Fig. 4 (see definition 12 in Appendix A for a precise definition of meta-model triple). The upper part (abstract syntax) of the meta-model triple depicts a slight variation of the UML 1.5 standard meta-model proposed by OMG for sequence diagrams[3] [31]. The lowest meta-model in the figure declares the concrete appearance concepts and their relations. Its elements are in direct relationship with the graphical forms that will be used for graphical representation. Abstract class *ConcreteElement* has two abstract edges *AbsMessage* and *AbsLifeLine*. *ConcreteElement* has three children: *StartPoint*, *ActivationBox* and *Object*. *Message*, *StartMessage* and *createMessage* refine abstract edge *AbsMessage*. They restrict the kind of *ConcreteElement* types that can be connected through a message: *StartPoint* and *ActivationBox*, *ActivationBox* with itself and *ActivationBox* and *Object*. A similar situation happens for *AbsLifeLine*.

The correspondence meta-model specifies the possible relations between elements of the concrete and abstract syntax. This is done by means of classes *CorrespondenceMessage* and *CorrespondenceObject*. The correspondence functions for the former node go to *Stimulus* and *AbsMessage*. As the latter is an abstract edge, this means that correspondence nodes with type *CorrespondenceMessage* can have correspondence functions to each one of the *AbsMessage* children edges. Note that including the *AbsMessage* abstract edge simplifies the correspondence graph. Otherwise we would need three node types in the correspondence graph, to relate *StartMessage*, *createMessage* and *Message* links with *Stimulus* objects.

Next subsection shows how ATT-graphs can be rewritten by means of TGG rules.

3.2 Attributed Typed Triple Graph Transformation with Inheritance

This section presents the basic concepts of attributed typed triple graph transformation in the DPO approach in an intuitive way. See Appendix A.2 for an introduction to the basic theory, and [23] for a complete presentation. In [27] TGGs are defined following the single pushout [13] (SPO) approach and are restricted to be monotonic (its LHS must be included in its RHS). Here we use the DPO approach and do not take the restriction of monotonicity. Moreover, we allow rules to have application conditions.

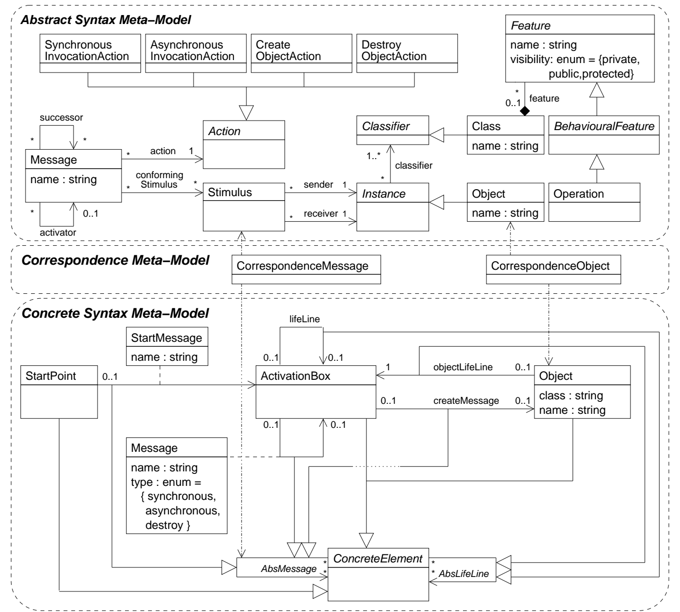

Fig. 4 A Meta-Model Triple Example.

The main idea in the DPO approach is that rules are modelled using three components: L, K and R. The L component contains the necessary elements to be found in the graph (called *host graph*) to which the rule is applied. K (the kernel) contains the elements that are preserved by the rule application. Finally, R contains the elements that should replace the identified part in the structure that is being rewritten. Therefore, $L - K$ are the elements that should be deleted by the rule application, while $R - K$ are the elements that should be added. In the DPO approach, graph transformation is formalized using category theory. In this way, not only graphs can be rewritten but objects in any (weak) adhesive HLR category [14] such as graphs, hypergraphs, Petri nets or triple graphs (see the end of Appendix A.1 and [23]). In our case, L, K and R are ATT-graphs. In the figures of the paper we omit the K component, and elements in L and R are labelled with numbers. Elements having equal numbers in L and R are preserved by the rule, and thus belong to K. The upper part of Fig. 5 shows a triple rule that connects one object and its classifier in the abstract graph.

A triple rule can be applied to a host ATT-graph if an occurrence (a match) of the rule's left hand side (LHS) is found in it. If such occurrence is found, then it can be substituted by the rule's right hand side (RHS). Such rule application is called *direct derivation*. Fig. 5 shows an example of direct derivation, in which a TGG rule is applied to ATT-graph G, yielding graph H (written $G \Longrightarrow H$). Match m identifies the elements of the rule's LHS in G, and the occurrence is depicted using numbers.

[3] This has been changed in the 2.0 version of UML, as now the concept of *LifeLine* is part of the meta-model. However, similar problems remain for the ordering of messages. Moreover, sequence diagrams have become more complex with the inclusion of combined fragments.

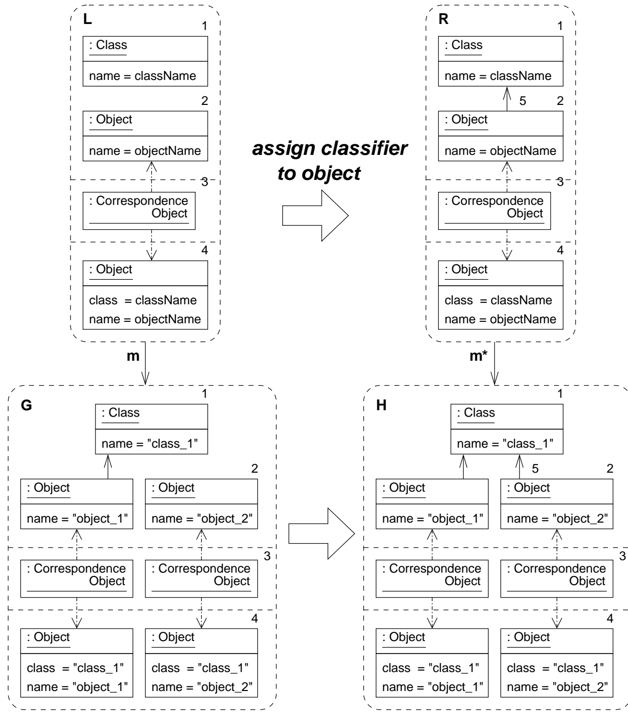

Fig. 5 A Direct Derivation Example.

In order to apply a rule, the DPO approach requires two additional conditions. The first one is known as "dangling edge" condition [13] and forbids rule application if deleting a node causes some edge to become dangling (i.e. the deleted node is the source or target of an edge in the host graph, and the edge is not explicitly included in the rule's LHS). The second requirement is called the "identification condition", and states that if two different elements in the LHS are identified by the match (through a non-injective matching) then they should be preserved by the rule.

A triple graph grammar (TGG) is made of a set of triple rules and an initial ATT-graph $TriAS$. The language generated by the grammar are all possible ATT-graphs derived from zero or more applications of the rules in the set starting from $TriAS$, written $L(TGG) = \{TriTAG | TriAS \Rightarrow^* TriTAG\}$.

In order to avoid creating twice the link between the object and the classifier, the rule in Fig. 5 should also check that the link has not been created before. This kind of negative test can be done by providing rules with application conditions, which further restrict rule applicability. One of the most common kinds of application conditions are *negative application conditions* (NACs). They consist of an extra ATT-graph (related to the LHS) that should not be present in the host ATT-graph (related to the LHS occurrence) for the rule to be applied. Fig. 6 shows two rules with NACs. The first rule creates an object in the abstract syntax (label 6) if an object has been created in the concrete syntax. The rule cannot be applied if the object in the concrete syntax (label 1) is already related to an abstract syntax object. This additional condition is tested with the NAC. The second rule connects an object with its classifier in the abstract syntax. The rule cannot be applied if they are already connected. The latter rule is in fact the complete version of the one shown in Fig. 5.

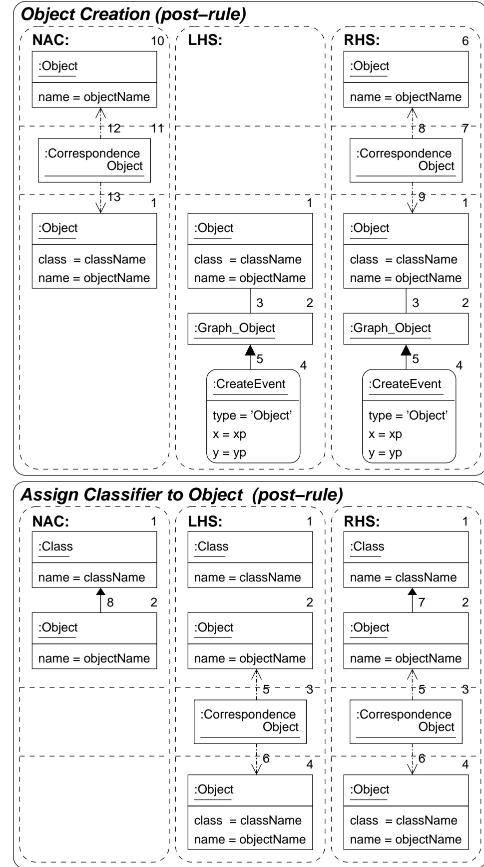

Fig. 6 Example of TGG Rules with NACs.

In this paper we also use a more complex kind of application conditions, made of an ATT-graph X and a set of ATT-graphs Y_j (see definition 11 in Appendix A). In this case, a rule can be applied if, given a match of the LHS, if an occurrence of X is also found, then an occurrence of some Y_j should also be found. NACs are a particular case of this kind of conditions where the Y_j set is empty.

In order to benefit from the inheritance structure of the meta-model triples, we allow triple rules to contain instances of abstract classes ("abstract objects") in the LHS (following a similar approach to [3,16], but for triple graphs and considering also edge inheritance). Of course a host ATT-graph cannot contain abstract objects. However, abstract objects (and in general any object whose classifier has children) in the rule's LHS can be matched to instances of any subclass of the abstract object classifier. We call this kind of rule inheritance-extended triple rules, or IE-triple rules. This kind of rules are indeed equivalent to a number of concrete rules, resulting from

the valid substitutions of each node and edge in the IE-triple rule by all the concretely typed nodes and edges in its inheritance clan (i.e. subnodes and subedges). If the set of equivalent rules of an IE-triple rule has cardinality greater than one, the IE-triple rule is called *IE-triple meta-rule*. Therefore, this kind of rules allows expressing computations in a more compact way than regular TGG rules. The application of an IE-triple meta-rule is equivalent to the application of one of its concrete rules (see [23] for details). Nodes and edges abstractly typed are thus allowed to appear in the LHS of an IE-triple rule. However, if an abstract node appears in the RHS, then it must also appear in the LHS. That is, we do not allow creating elements with an abstract typing. This could be done in principle, and the meta-rule would be equivalent to a number of concrete rules resulting from the substitution of the elements with abstract types by elements with concrete one in the inheritance clan. However, this could result in non-determinism when applying the meta-rule, which we want to avoid. See definition 15 in Appendix A for a formal definition of IE-triple rule.

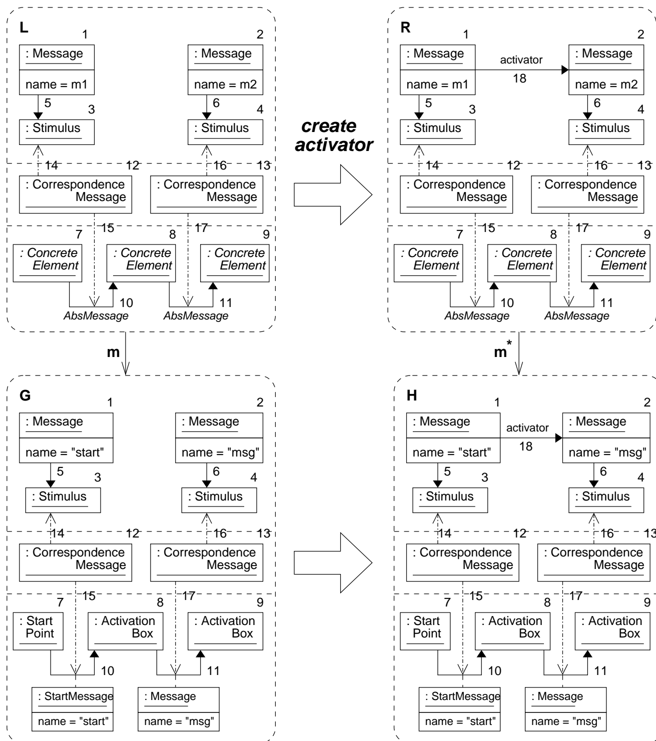

Fig. 7 An Example of IE-Triple Meta-Rule and Derivation.

The top row of Fig. 7 shows an IE-triple meta-rule example. The rule identifies the activator message of another one, creating an edge in the abstract graph (the rule is simplified, we do not include application conditions for clarity). Nodes 7, 8 and 9 and edges 10 and 11 of the concrete graph have an abstract typing. The meta-rule is equivalent to four concrete rules. Node 7 can take types *StartPoint* or *ActivationBox* in the concrete rule, node 8 has to be an *ActivationBox*, and node 9 can be an *Object* or an *ActivationBox*. Thus, four combinations are possible, where the edge types are determined by the choice of node types. The figure also shows a direct derivation example, where abstract elements 7, 8, 9, 10 and 11 in the rule take concrete types *StartPoint*, *ActivationBox*, *ActivationBox*, *StartMessage* and *Message* in the triple graph G.

Once we have defined the basic concepts regarding triple graphs and triple rules, next section presents event-driven grammars.

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 editing operations in AToM^3 [10], although the approach can also be applied to other tools. The actions a user can perform in AToM^3 are *creating*, *editing*, *deleting* and *moving* an entity or a link, 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 user events in the rules. This is very different from the *syntax directed* approach, where graph grammar rules are defined for VL generation and 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. The events that the user generates may trigger the execution of some rules. In this work, rules are IE-triple rules and are used to build the abstract syntax model, to perform consistency checking and for concrete syntax layout.

An event-driven grammar contains three sets of predefined rules. The first one, called *event-generator rules* (depicted as *evt* in Fig. 8) models the generation of events by the user. Another set of rules (*action rules*, depicted as *sys-act* in Fig. 8) models the actual execution of the event (creating, deleting entities, etc.). Finally, an additional set of rules (called *consume rules*, depicted as *del* in Fig. 8) models the consumption of the events once the action has been performed. In addition, the VL designer can define his own rules to be executed after an event is generated by the user and before the execution of the *action rules* (depicted as *pre* in Fig. 8), or after the *action rules* and before the *consume rules* (depicted as *post* in Fig. 8). 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 actions, which is similar to a post-condition. The working scheme of an event-driven grammar is shown in

Fig. 8. All the sets of rules (except the event-generator rules in *evt*, which just produce a user event) are executed as long as possible (note the asterisk on the derivation arrow).

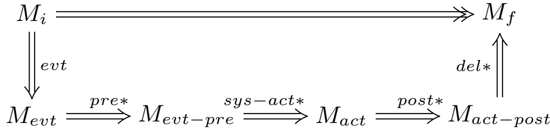

Fig. 8 Direct Derivation of Event-Driven Graph Grammar.

All the models in Fig. 8 M_i, M_{evt}, $M_{evt-pre}$, M_{act}, $M_{act-post}$ and M_f are attributed triple graphs typed by a meta-model triple. However, the sets of rules *evt*, $sys - act$ and *del* are restricted to modify the concrete graph only, which represents the concrete syntax. On the other hand, rules in *pre* and *post* are unrestricted IE-triple rules, which can be used to propagate the changes due to user events to the abstract syntax model (abstract graph). A direct derivation by an event-driven grammar (caused by a user event) is depicted as $M_i \Longrightarrow M_f$. The formal definitions of event-driven grammar and derivation are given in definitions 16 and 17 in Appendix B.

Fig. 9 shows the AToM^3 base classes for the concrete syntax. We already showed some of these classes in the second meta-level of Fig. 2. As stated before, all concrete syntax symbols inherit either from *Entity* (if they are icon-like entities) or from *Link* (if they are arrow-like entities). 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 *LinkSegment* objects. These can go either from *Entities* to *Links* (*e2l*) or the other way round (*l2e*).

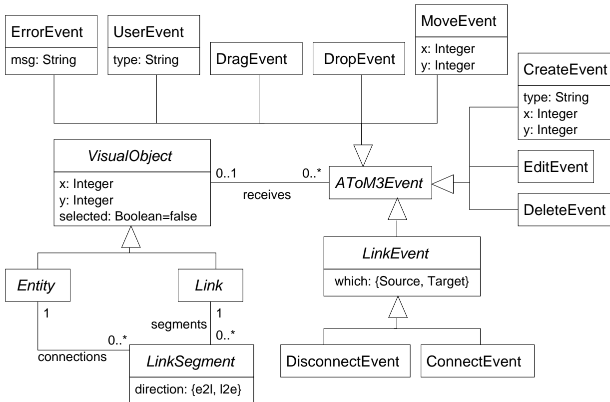

Fig. 9 AToM^3 Base Classes for Concrete Syntax Objects and User Events.

Abstract class *ATOM3Event* in Fig. 9 models the events that can be generated by the user, and can be associated to a *VisualObject*. Some concrete events have additional information, such as *CreateEvent*, which contains the type of the *VisualObject* to be created and its position. *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 another one associated to the target. *ErrorEvent* signals an error associated with a certain object, in such a way that AToM^3 presents the text of the error and highlights the object. Finally, the *UserEvent* class can be used to define new events.

From now on, we assume that the classes in Fig. 9 (together with classes *ASGNode* and *ASGConnection*, see Fig. 2) are the base classes for the concrete syntax graph of meta-model triples, in a similar way as in the second meta-level of Fig. 2. In the following, we present some event-driven rules for the *evt*, $sys - act$ and *del* sets. They model the behaviour of the AToM^3 tool. As the correspondence and abstract graphs are empty in these rules, we omit them and show only the concrete graph.

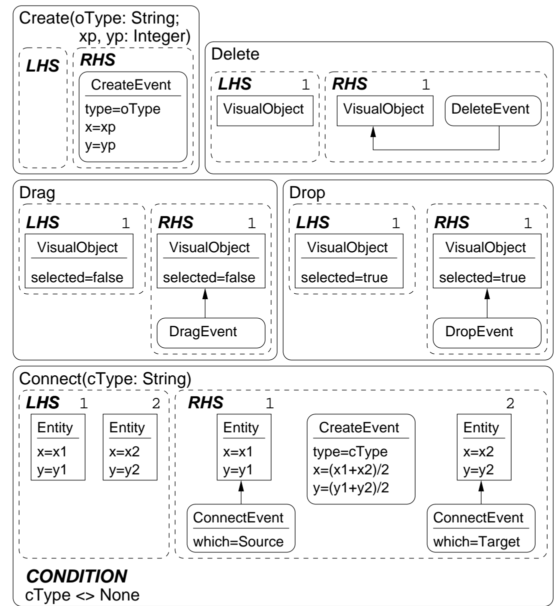

Fig. 10 Some of the *Event-Generator* Rules.

Fig. 10 shows some of the *event-generator* rules (depicted as *evt* in Fig. 8), 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. 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. On the other hand, if the entities cannot be connected, then the type is empty (*None*), and the rule cannot be executed (see the application condition). Note that all these are meta-rules, as we do not care about the exact type of the graphical elements. That is, these rules are general, valid for any VL.

For simplicity, the event-generator rules presented in this paper are triggered by a specific sequence of user actions (e.g. by clicking a button and then the canvas for the *Create* rule). However, in [4], it is shown how the trigger action can be made explicit in the rules in order to handle different interaction possibilities (e.g. the selection of a menu option instead of a button click). Supporting more complex interaction patterns as triggers is up to future work.

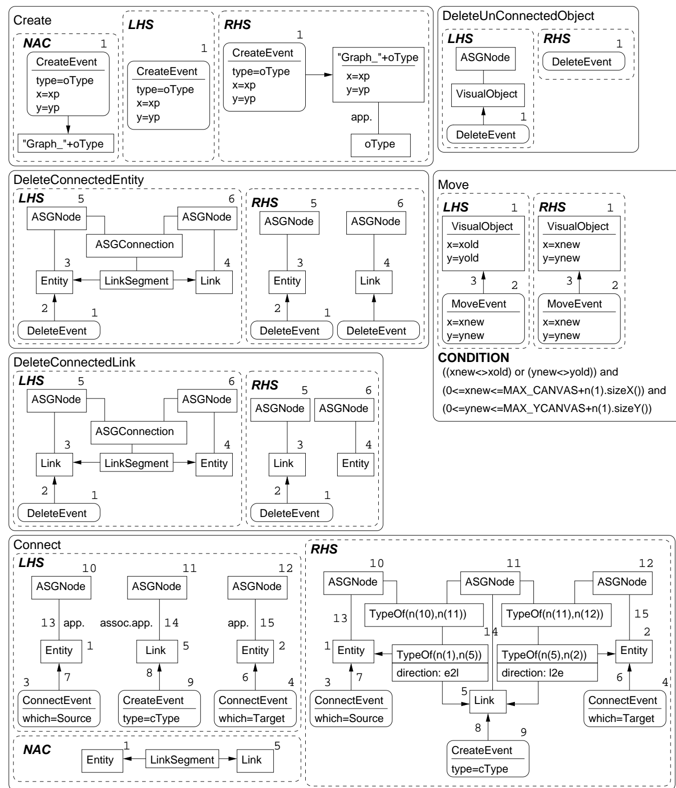

Fig. 11 Some of the *Action* Rules.

Fig. 11 shows some of the rules that model the real execution of the events (depicted as *sys-act* in Fig. 8). The first rule models the actual creation of an instance (subclass of *ASGNode*, see Fig. 2), together with its associated visual representation (whose type name is the same as the non-visual instance, but starting by *"Graph_"*). Three of the following rules model the execution of a delete event. In the first case (*DeleteUnConnectedObject* rule) the object has no connections. The rule is not applicable (due to the DPO dangling condition[4]) if the *ASGNode* to be deleted has any connection. 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, which erases one of the associated segments. Please note that all the rules are executed as long as possible (see Fig. 8). Therefore, when no more segments are connected to the link, the link itself is deleted by rule *DeleteUnConnectedObject*, which can delete *Entity* and *Link* objects (as they are subclasses of *VisualObject*).

The *Connect* rule models the connection of a link to two entities. Rule *Connect* in Fig. 10 generates a *CreateEvent* for the link. In this way rule *Create* in Fig. 11 is executed first, creating the link with the correct type. Next, rule *Connect* in Fig. 11 can be applied, as classes *Entity* and *Link* are the base classes for all graphical objects. 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. The function takes two objects as arguments and returns the type of the object that can connect them.

The *Move* rule simply modifies the position attributes of the object, in case the new position is valid in the canvas. The *Move* event does not need to be propagated to the adjacent elements, as we have modelled links and entities to be connected through segments, and these do not hold position information, but are drawn from the link to the entity. However, a propagation of the move event from entities to links could also be modelled (by adding several extra *pre-* and *post-* rules) if useful for certain VLs, or if we want to model a multiple selection and then moving several graphical objects at the same time. In fact, being able to express different tool behaviours in a flexible way was one of the goals of the event-driven grammars approach. In the same way, AToM^3 allows object overlapping. However, it could be possible to forbid object overlapping by providing a negative application condition for the *Move* rule. This NAC would contain one *VisualObject* placed in a position that overlaps with the new position of the object being moved. Modelling other spatial relationships, such as containment or adjacency is also possible. This could be done by means of post-rules created by the VL designer. The rules might check that, when an object of some specific type has been moved, all the connected objects of a certain type should also be moved, in order to maintain them adjacent. An example of this kind of rules is presented for sequence diagrams in section 5.3, which shows how, when moving an object, all its activation boxes are moved as well. These mechanisms to deal with spatial relations between

[4] which forbids rule application if deleting a node produces dangling edges, see [13].

graphical elements could be generalized by extending the AToM^3 meta-metamodel with special relations (in the style of [5]) between *VisualObjects*, and adding the corresponding *action* rules. This is up to future work.

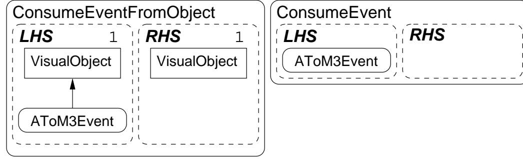

Fig. 12 All the *Consume* Rules.

Finally, a last set of predefined rules (shown in Fig. 12, and depicted as *del* in Fig. 8) models the deletion of the events. Rule *ConsumeEventFromObject* deletes any event that is connected to a *VisualObject*. Rule *ConsumeEvent* deletes any unconnected event. Again, this rule cannot be applied if the event is connected due to the dangling edge condition.

Fig. 13 shows an example with a derivation sequence. We use the meta-model triple of Fig. 4, which defines the abstract and concrete syntax of sequence diagrams. Some of the post-rules for the example are shown in Fig. 6. Moreover, we assume the meta-model triple has been created with AToM^3. Thus, the elements in the concrete part of the meta-model triple in Fig. 4 inherit from the AToM^3 base classes. Therefore they can receive events, according to the meta-model in Fig. 9. The resulting concrete graph of the meta-model triple is very similar to the one found in the second meta-level of Fig. 2.

The example starts with an empty concrete syntax and assume there is an already defined class in the abstract syntax (created by a class diagram). We model the creation and editing of an object by the user. In the first step, the user generates a creation event by clicking on the "create object" button of the user interface and then on the canvas (at coordinates $(10, 10)$). Thus, a *CreateEvent* object appears in the concrete syntax. As there is no applicable "pre-" rule, the "sys-act" rules can be applied. These rules implement the event semantics, and therefore an object is created in the concrete syntax in step 2. No additional "sys-act" rule is applicable; consequently the grammar execution enters in the "post-" rules step. Here, rule "Object Creation" (shown in Fig. 6) can be applied. In this way, in the third step an object is created in the abstract syntax, linked to the object in the concrete syntax by a correspondence object. No additional "post-" rule is applicable and the execution enters in the "del" step. Thus, in step four, the *CreateEvent* is deleted.

In step five, we model a user editing action on the previously created object. This is performed by clicking on the "Edit" button of the user interface and then on the visual object to be edited. Thus, an *EditEvent* object

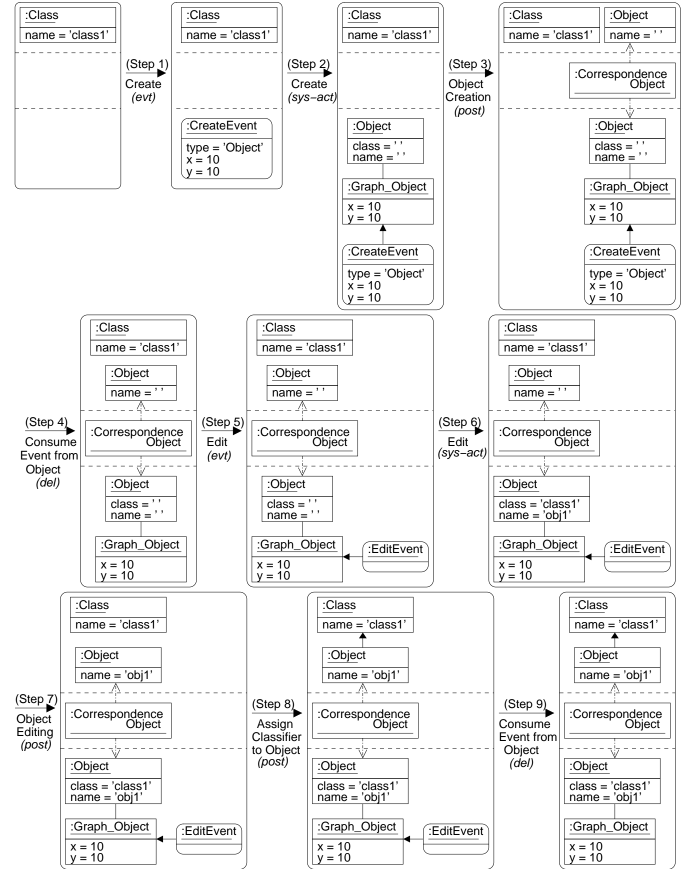

Fig. 13 A Fragment in the Execution of an Event-Driven Grammar.

is created and connected with the selected visual object. In step six, we model the execution of such editing action. Thus, the value of the attributes (*name* and *class*) changes in the object associated to the visual representation. Then a "post-" rule is triggered, that modifies the *name* attribute of the abstract syntax object. Moreover, in step eight, an additional "post-" rule ("Assign Classifier to Object" in Fig. 6) is executed that associates the object with its classifier at the abstract syntax. Finally, the execution reaches the "del" step, where rule "Consume Event from Object" is fired. The rule erases the *EditEvent* object.

5 Example: Sequence Diagrams

As an example of the developed techniques, we describe an environment to define the abstract and concrete syntax of UML sequence diagrams. We use the meta-model triple in Fig. 4, and assume – as in the previous example – that the elements of the concrete graph of the meta-model triple inherit from the AToM^3 base classes. Starting from this triple meta-model, AToM^3 generates a tool where the user can build models according to the concrete 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 event-driven rules models specific spatial relations between the elements drawn in the concrete syntax layout (such as the automatic alignment of the activation boxes that belong to the same object). Finally, a set of triple rules checks the consistency between the sequence diagram and existing diagrams. The three different sets of rules are presented in the following subsections; but first, we briefly present our approach to the visual modelling of systems with multiple views (such as UML).

5.1 Multi-View Modelling

In order to cope with the complexity of system modelling, one may have to use partial views for the specification of their different aspects (structure, behaviour, etc.). A different modelling language is usually used for the specification of each view. This is the case of UML [31], where system structure and behaviour can be described using several modelling notations. Although these notations can be independently used, they were defined and related by a single meta-model, to complement each other. Thus, one can refer to the same concept in different diagrams, and model different aspects of a given entity. For example, a class may appear in several class diagrams, it can be assigned a statechart, and then be referenced as the classifier of a number of instances in object or sequence diagrams.

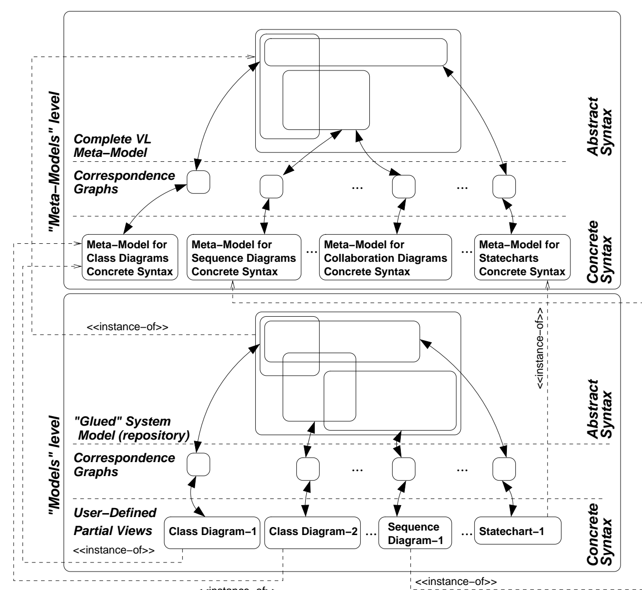

Fig. 14 Modelling a Multi-View System with UML (instance-of relations of correspondence graphs are omitted for clarity).

This situation is depicted in Fig. 14. The figure shows how the user can model a system using several views and the different concrete syntaxes of the views at the "Models" level. Each view is specified with a diagram type, using a given concrete syntax, and being conformant to its corresponding meta-model. The latter is represented with the relation *instance-of* in the figure. In addition, each view has also an associated abstract syntax. The different views are related at this abstract syntax representation. In fact, it is possible to obtain a unique abstract syntax model (called *repository*), which is the result of "gluing" the abstract syntax of the different views. Thus, fragments of the abstract syntax obtained from each concrete syntax view may overlap. Methods should be provided to connect and assure consistency between the different views at the abstract syntax level. Here we propose using triple graph transformation for this purpose. We use it in combination with event-driven grammars to build the abstract syntax model. Moreover, we can use it for consistency checking, in order to make sure that the new elements added by a concrete syntax view are consistent with the already existing elements at the abstract syntax. An example is shown in subsection 5.4. Note that in order to build an environment for UML with this approach, at the abstract syntax one should provide the full UML meta-model, while at the concrete syntax different meta-models for each of the diagram types should be given. Fig. 4 presented only the abstract syntax of a small part of the UML meta-model (together with its concrete syntax) relevant for sequence diagrams.

5.2 Abstract and Concrete Syntax of Sequence Diagrams

We have defined the abstract and concrete syntax of sequence diagrams using the meta-model triple in Fig. 4. In this subsection, we provide event-driven rules to construct the abstract syntax from the user actions at the concrete level. These rules manage the creation, editing and deletion of *Objects*, the creation, editing and deletion of *Messages*, and the creation and deletion of *Life Lines*. The graphical actions that do not change the abstract syntax (like creating an *Activation Box* or moving an element) do not need the definition of extra rules apart from the ones provided by AToM^3 (Figs. 10, 11 and 12). In addition, we have provided some rules for layout management, which are shown in section 5.3.

Rules for the creation, editing 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) are shown in Fig. 6. The top-most rule creates the object at the abstract syntax level, while the rule below connects (at the

abstract syntax level) the object with its corresponding class. If the second rule cannot be applied, it means that such class has not been created in any class diagram yet. 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 some rules to check and signal inconsistencies. These rules are explained in subsection 5.4 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-* action rule that erases the *delete* event on an object and presents an error message if it has some connection. This is the main idea of *pre-* action rules: checking if some condition is not met, and in that case, inhibiting the event execution by deleting the event itself.

The creation of a message is equivalent to connecting two elements belonging to the concrete syntax (*ConcreteElement*, see Fig. 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 *ActivationBoxes* by means of a *Message* relation, the connection of a *StartPoint* to an *ActivationBox* by means of a *StartMessage* relation, and the connection of an *ActivationBox* to an *Object* by means of a *createMessage* relation. The event-driven rules for managing the first two concrete cases are very similar. That is, we should have a first rule to create a *Message* relationship if its source and target are activation boxes and another similar one except for the relationship type (*StartMessage*) and its source (*StartPoint*). Since the two rules have the same structure, we use instead the IE-triple meta-rule shown in Fig. 15. The rule generates the abstract syntax of a new message created by the user, adding a relation between the concrete syntax of the new message 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 rule has to process the relations between the newly created abstract syntax message and the rest of the abstract syntax model. That is, the successor, predecessor and activator messages of the created one have to be computed (relations *successor* and *activator*, see Fig. 4), as well as the objects sending and receiving the message (relations *sender* and *receiver* in the same figure). Additionally, we have to check if the new message activates in its turn another block of messages. To make easier such complex process, we have broken down the creation event in a set of six user-defined events, each performing one step. The user events are created in the RHS of the rule, and are processed by some additional rules. Processing a *createMessage* cannot be included in this meta-rule, as in the abstract syntax a *CreateObjectAction* object should be created, whereas in the meta-rule of Fig. 15 a *SynchronousInvocationAction* object is created for both *Messages* and *startMessages*. Therefore an additional, similar rule is needed for processing *createMessage* objects.

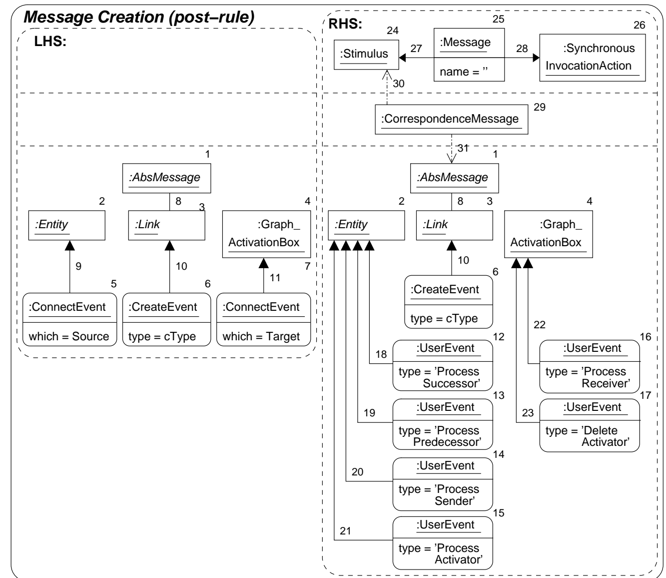

Fig. 15 Meta-Rule for Creating *Messages* and *startMessages*.

Fig. 16 shows the rule for processing the user event "Process Successor". The successor of a message is the next message in the same activation block of an object. An activation block is made of one or more *ActivationBox* objects linked through life lines. The first activation box has an incoming message, and the rest have at most one outgoing message and no incoming messages. The activation block is visually represented by gluing together all the activation boxes. Note that the user event is associated to the activation box which is source of the message. Thus, given a message for which its successor has to be calculated, the rule searches for a message going out from the next activation box in the same life line. The next activation box of "1" in the rule's LHS is the object labelled with "8", and its outgoing message is labelled with "11". In addition, the second activation box has to be in the same activation block, that is, it cannot be the beginning of a new activation block (i.e. it cannot receive an incoming message). This is checked by NAC2, which forbids an incoming message to the second activation box. Finally, the NAC1 checks that a *successor* relation does not exist yet. If all these conditions are met, the rule creates a *successor* relation (label 36) between the first message and the following one at the abstract syntax, and deletes the user event. The rule for handling the user event "Process Predecessor" is very

similar to this, but in this case the previous message in the same block of activation is searched, not the next.

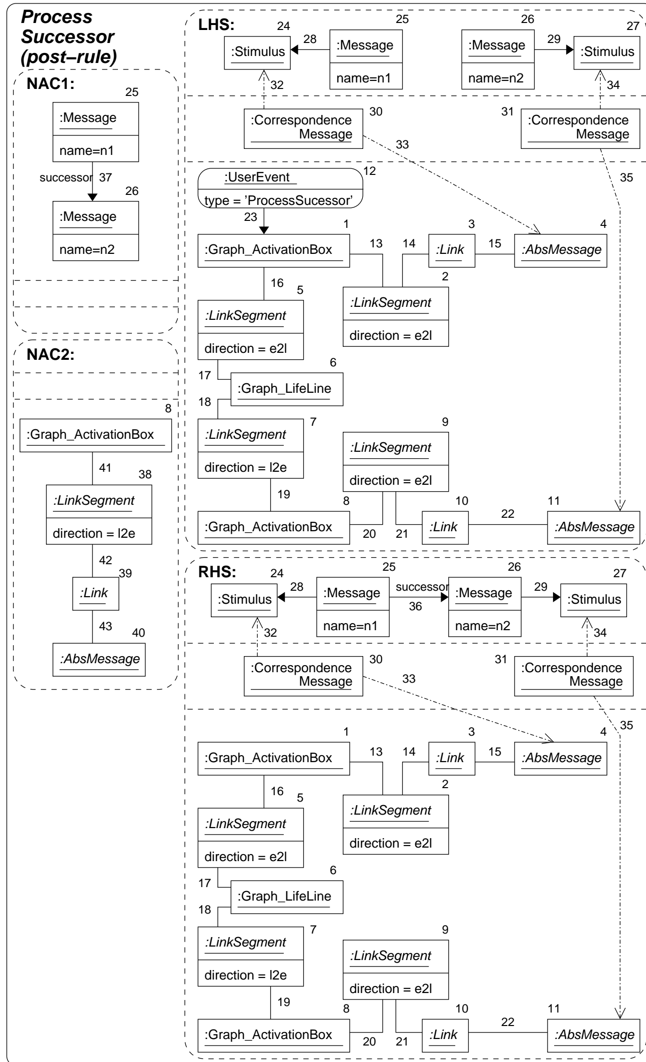

Fig. 16 Meta-Rule for Assigning Message Successors.

A total of 15 rules have been defined to manage the creation, editing and deletion of *Objects* and *Messages*. Some other rules, similar to the previous ones, manage the creation and deletion of *lifeLines*. In this way, some of the user events shown in Fig. 15 (and their corresponding rules) have been reused. Thus the number of rules has been highly reduced. Due to space limitation, we do not show all the rules, which are 38 in total.

Note that starting from the meta-model triple, it could be possible to automatically generate a skeleton for some of these rules. Full generation for all the rules was achieved in [22], but in the restricted case in which the concrete graph of the meta-model triple is a restriction (a subset) of the abstract syntax part of the meta-model, and thus the relation between concrete and abstract elements is one-to-one. The problem we are dealing with in this article is more general as we do not have such restriction. However, we can generate a skeleton for the creation, editing and deletion rules. We just have to generate such rules for each connected elements in the concrete and abstract meta-models through a correspondence node. Nonetheless, rules generated in this way are not fully complete as in general we cannot know how the attribute mapping is done. Moreover, often not only an element should be created in the abstract syntax, but it also has to be connected to other elements (as rule *Assign Classifier to Object* in Fig. 6 does). In addition, sometimes these skeletons should also be completed with additional elements as, for example, an element in the concrete syntax may be related to more than one element in the abstract syntax (as in the case of rule *Message Creation* in Fig. 15).

5.3 Concrete Syntax Layout

Event-driven grammars can also be used to model the behaviour of the tool in the concrete syntax layout. They can help in handling complex spatial relations between the elements in a model, such as adjacency, containment or alignment. Obviously, in these cases rules should only modify the concrete syntax model, although its application could be restricted by certain conditions taking into account both abstract and concrete syntax elements. As an example, Fig. 17 shows a couple of rules to maintain aligned in the same vertical line all the activation boxes corresponding to the same object (that is, those related through a life line relation).

The first rule, “Aligned Life Lines Connection”, is a pre-rule. It will be tried when connecting an object with its first activation box through an *objectLifeLine* relation, or two activation boxes of the same object through a *lifeLine* relation. In order to be able to treat both cases, the LHS of the rule contains the abstract class *Entity* (labelled as “1”) that is source of the new relation. This can be matched either to an *Object* or to an *ActivationBox* concrete object. Any other possible matching is forbidden by the condition, which prevents the application of the rule when the creation event is neither for an *objectLifeLine* nor for a *lifeLine* relation. If the rule is applied, the life line relation and the target activation box are aligned in the same x coordinate as the source entity (centered in the middle of the width of the source entity).

The second rule, “Aligned Life Lines Movement”, is also a pre-rule. Its purpose is to maintain aligned an object with all its activation boxes when the user moves any of them in the concrete syntax. Thus, the LHS of the rule detects entities related through a life line, where one of them has received a *MoveEvent*. The abstract class *AbsLifeLine* (labelled as “3”) can be matched to both concrete *objectLifeLine* and *lifeLine* links. In this way, we compress in only one rule the cases of moving an object and moving an activation box. The RHS of the

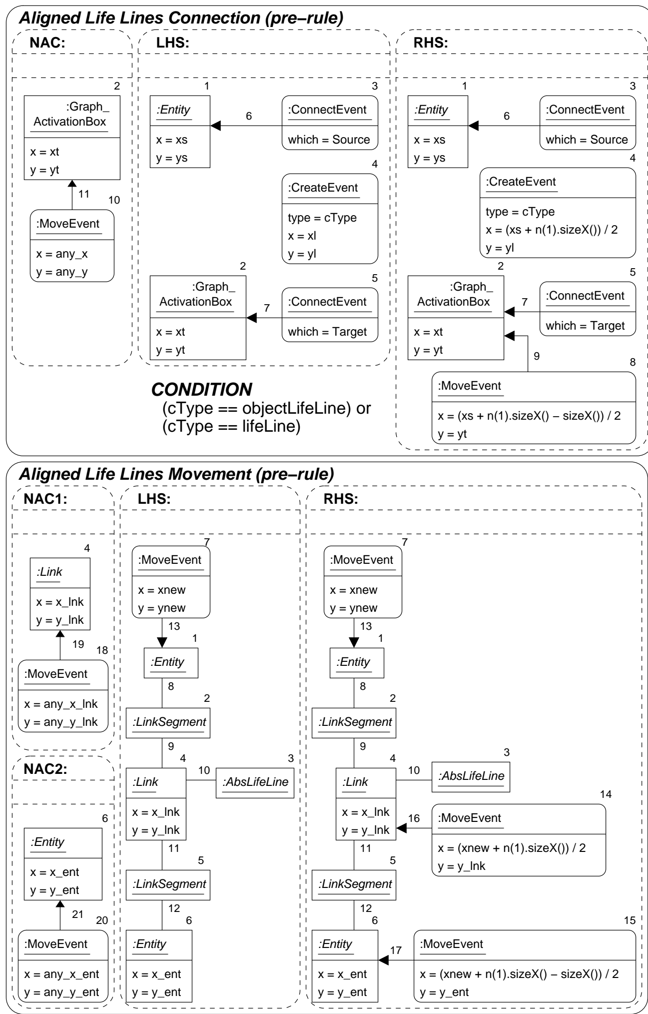

Fig. 17 Rules for the Alignment of Life Lines.

rule propagates the *MoveEvent* properly to the related link and entity. The rule can be executed in an iterative way (as long as possible), so the event is propagated to all the activation boxes in the same life line. Besides, since the rule does not restrict the link segment direction (*e2l* or *l2e*), the event will be propagated up and down the row of activation boxes. That is, it does not matter whether the entity that received the *MoveEvent* is source or target of the life line relation. Finally, the NACs forbid applying the rule twice to the same entity. Note that this rule will also be tried after applying the rule "Aligned Life Lines Connection". In this way, if the activation box source of the connection has a row of already connected activation boxes, all of them will get aligned with the newly connected element.

Additional layout rules control the adjacency (in the vertical direction) of all the activation boxes making an activation block. Note that for the running example we use a simplified concrete syntax meta-model for UML 1.5 sequence diagrams (see Fig. 4). In particular, we do not allow the branching of life lines, as the cardinality of the *lifeLine* relation is 0 or 1. However, this feature could be modelled with additional event-driven rules.

5.4 Consistency Checking

Triple rules can be used not only to maintain coherence between concrete and abstract syntax, but also to check consistency between different diagrams. The present work is part of a more general project with the aim to formalize the dynamic semantics of UML [20] by means of transformations (at the abstract syntax) into semantic domains (up to now Petri nets). Before translation, consistency checking should be performed between the defined diagram (in this case a sequence diagram) and existing ones, such as class diagrams.

As stated before, while the user builds the concrete syntax of a sequence diagram, some event-driven rules add abstract syntax elements to a unique abstract syntax model (the *repository*). 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 triple rules we can perform consistency checking between the sequence diagram and the existing abstract syntax model generated by previously defined diagrams. For example, we may want to check that the classes of the objects used in a sequence diagram have been defined in some of the existing class diagrams; that if an object invokes a method of another object, the method should have been defined in the class of the invoked object; and in addition, that such invoked method should be visible from the calling class and there should be a navigable relationship between both object classes.

We have defined *consistency triple rules* in such a way that their LHSs contain conditions sought in the defined diagram (a sequence diagram in our case), possibly in both the concrete and abstract parts. They contain application conditions as patterns to be sought in the complete abstract syntax model with which we want to check consistency. If the rule is applied, its RHS sends an event of type *ErrorEvent* to some of the concrete objects matched by the LHS. As an example, Fig. 18 shows a couple of consistency triple rules. Rule "Check Classes" displays an error if the class specified for an object in the sequence diagram has not been defined in some class diagram, so it is not present in the abstract syntax model (NAC). Note that this inconsistency was allowed during the modelling phase since we want flexible environments. However it is an error so it must be pointed out and fixed. Similarly, rule "Check Methods" displays an error if the classifier of the object that receives a method invocation does not define the corresponding *Operation*. This is modelled in the NAC (i.e. the rule cannot be applied and therefore the error is not given if the class defines such operation). Moreover, if the visibility of the operation is *private*, the classifier of the sender object should be the same as the classifier of

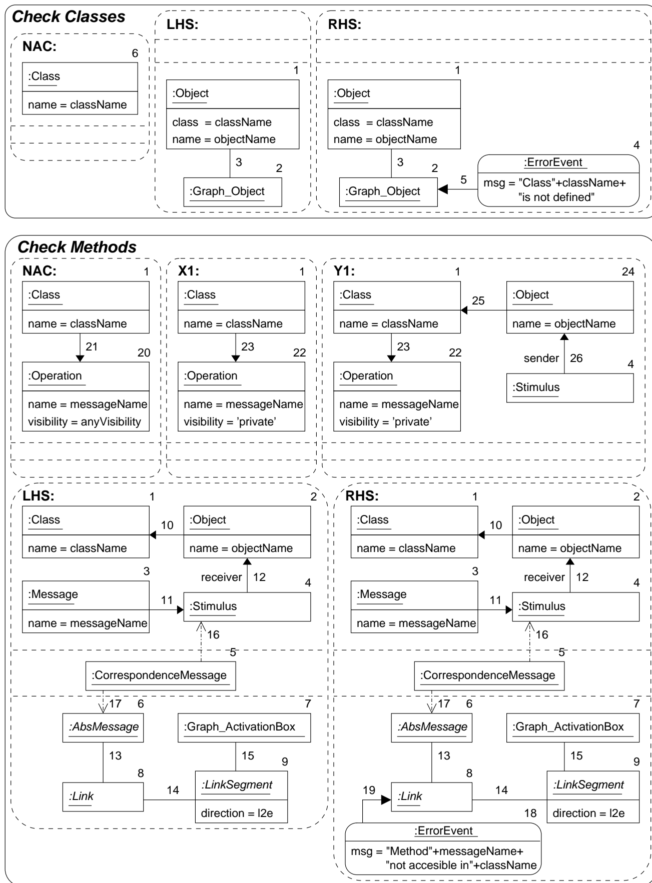

Fig. 18 Some Triple Rules for Consistency Checking.

the receiver object (labelled as "1"). This is modelled in the application condition ($X_1 \rightarrow Y_1$). Please notice that, at the concrete level, only a message ($AbsMessage$) and its target activation box ($Graph_ActivationBox$) are selected. Thus the rule is applicable to messages of types $createMessage$ and $Message$. That is, we do not want to consider $createMessage$ objects, as we assume each class has at least a default constructor.

6 Implementation in AToM^3

A prototype implementation of triple graph transformation was built in AToM^3. In this way, AToM^3 can now work either with regular graphs or with triple graphs. The user can define meta-model triples. In any component (abstract, concrete or correspondence) the user can edit a class diagram or load an existing one. Thus, regular meta-models can be reused in meta-model triples. From a meta-model triple, an environment is generated that allows the user to manipulate any of the three graphs. Nonetheless, the correspondence and the abstract graphs can be hidden to the user, who is then only able to use the concrete syntax elements.

Fig. 19 shows a picture of the generated environment from the meta-model triple in Fig. 4. The three components of the graph are visible and can be edited using the corresponding set of buttons to the left. However, the correspondence and abstract graphs are usually hidden, and user interaction takes place only in the concrete graph. In the picture, the concrete syntax graph shows a simple model where *msg0* is the starting message, which is received by object *object1*, and then sends message *msg1* to object *object2*.

With respect to the presented graph transformation techniques, AToM^3 already allowed the possibility to define graph grammar rules with the inheritance concept defined in [3]. Nonetheless, the tool did not allow the graphical modelling of application conditions. They had to be encoded as Python code. We have provided rules with application conditions as defined in [25]. Thus, they are available for regular and triple graph grammars. We have implemented the same scheme used in the paper to simplify the conditions. In this way, as the morphism from LHS to X (and from X to each Y_i) is total, if any element in the LHS (resp. X) does not have an image in X (resp. Y_i), it is copied and appropriately connected in the X (resp. Y_i) graph (but this is kept transparent to the user).

Fig. 20 shows a snapshot of AToM^3 being used to edit the TGG rules for consistency checking. In particular, one of the application conditions of rule *Check Methods* (the one labelled as Y_1 in Fig. 18). The main AToM^3 window is shown in the background. The dialog above is used to declare the rules of the grammar (in this case, named *CheckConsistencySD*) and shows two rules in the list. Note that with this dialog we can define a graph grammar or a parallel rule [9] amalgamating all the rules in the list. In the dialog above, rule *Check Methods* is being edited. Here we have specified that we want to use the inheritance concept (check button labelled as "Subtypes Matching"). Then, in the dialog above, we have declared two application conditions for this rule: "Exists Operation" and "Private Operation". The latter condition is being edited in the above dialog, where we have declared a "consequent graph" (the Y_i graph in the definition) named "Same Classifier". The next dialog is used to edit this graph. It has two buttons to edit the graph and the attribute conditions. Finally, the actual graph is shown at the bottom.

7 Related Work

At a first glance, the present work may resemble the *syntax directed approach* for the definition of a VL. In this approach a rule is defined for each possible editing action, and the user builds the model by selecting the rules to be applied. Our approach is quite different, as 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, and some events are produced as a result

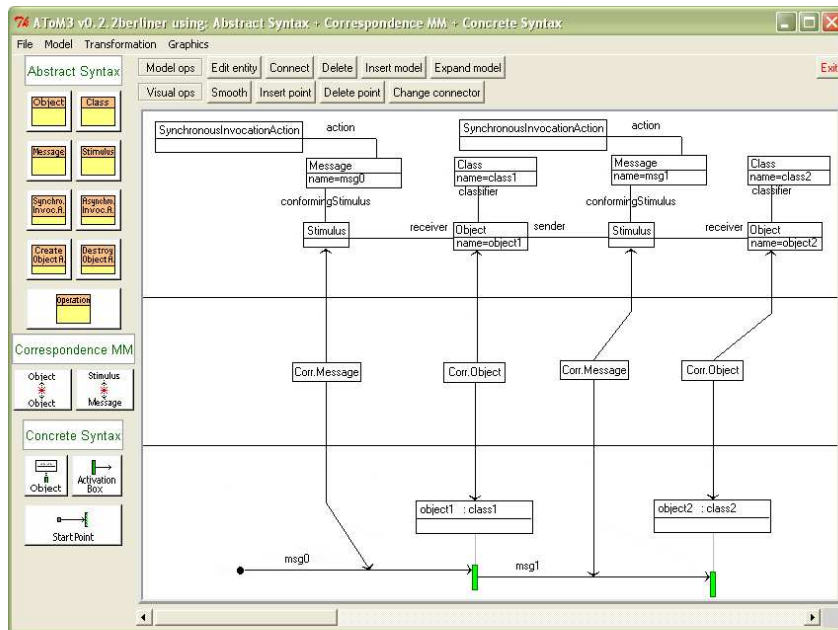

Fig. 19 Generated Environment with AToM^3 for Sequence Diagrams.

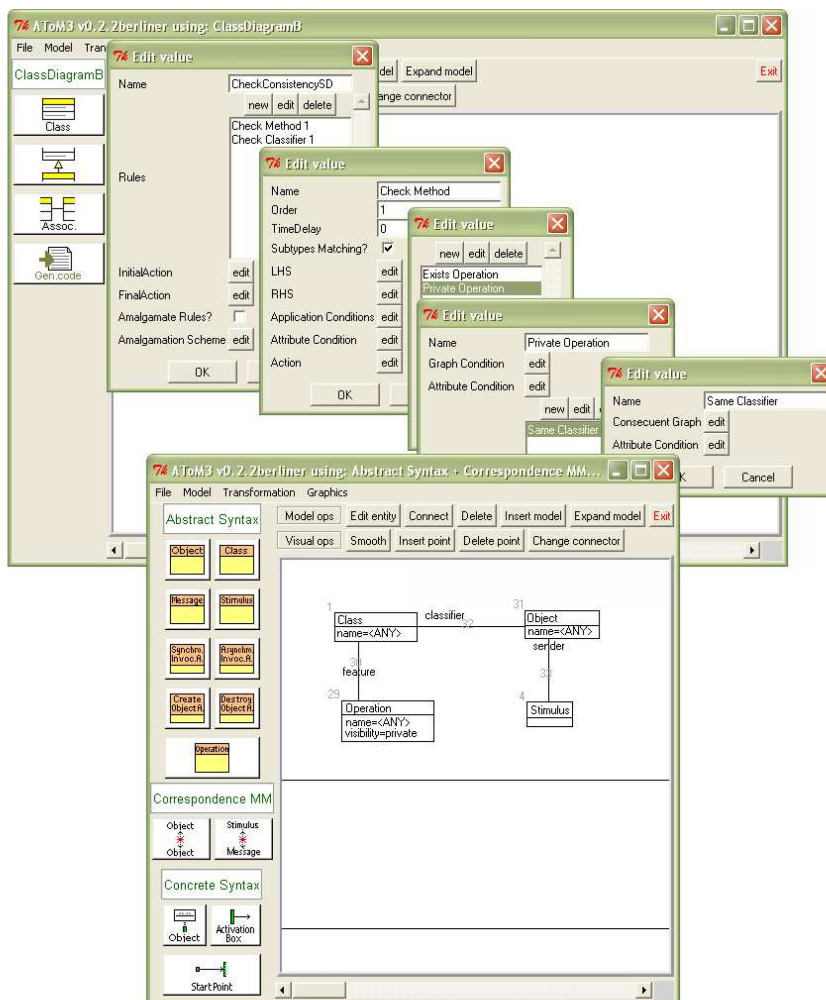

Fig. 20 Modelling the Application Condition of Rule *Check Methods*.

of such interaction. 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 graphs to build the abstract syntax model, to perform consistency checks and for layout management.

The present paper improves previous work of the authors. For example in [11], we proposed to build visual front-ends for OOCSMP (an object-oriented simulation language) using AToM^3, and concentrated in generating OOCSMP code from visual models (indeed using graph grammars). However, that work did not consider event-driven grammars, triple graph transformation or multiple views (however, we allowed attributes of entities to be models). Therefore in these previous works, the concrete syntax of the environment had to be in one-to-one correspondence with the abstract syntax. Moreover, the new concept of event-driven grammars allows the specification of much richer interaction possibilities with the modelling environment [4].

In the approach of [6], 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 had to be transformed into an HLG (using a regular graph grammar), and a parsing grammar had to be defined for the latter. Another parsing approach based on constraint multiset grammars is the one of CIDER [26].

In [5], a set of meta-models was identified for the definition of classes of VLs. The approach is based on a core meta-model with the basic elements regarding visualization. They extended this meta-model for different families of VLs by refining the graphical elements and by adding spatial relations (like containment) between them. Note how, our approach can be a complement to this idea, as we can define by means of rules the semantics of these spatial relations.

In [19], an approach to the rewriting of partial algebras and its application to VLs is presented. The idea is to have an internal algebra rewriting, and arbitrary external components. The abstract syntax of the VLs is transformed by rewriting rules (the internal algebra rewritings), while the concrete syntax layout is obtained using a constraint solver (the external component). Our approach is somehow more general in its application to VLs, as we do not restrict the abstract elements to be in one-to-one correspondence with concrete syntax elements. Moreover, we do not follow a syntax-directed approach, but represent interaction events explicitly, and these are handled by the rules.

Other approaches for the definition of environments for the different UML diagrams usually concentrate either on the concrete or the abstract syntax, but not on both. For example, in [7], graph transformation units are used to translate from sequence diagrams into collaboration diagrams. As both kinds of diagrams share the same abstract syntax, in our case a translation is not necessary, but we have to define triple rules to build the abstract syntax from the concrete one.

With respect to triple graph grammars, they were originally proposed in [27] as a means to derive lower-level, operational rules to perform forward or backwards translations, incremental updates or so called consistency observing analyzers. In the present paper, we provide a richer graph concept and a formalization of triple graph transformation in the DPO approach. However, the algorithms for derivation of operational rules for these richer graphs we propose are up to future work.

In the area of multi-view modelling, the *ViewPoints* approach [17] proposes a method for the integration of multiple perspectives in system development. A ViewPoint is an agent with partial knowledge about the system and has a style (the used notation); a domain (the area of concern); the actual specification and the work plan (available actions to build the specification). In particular, two of the work plans are *In-* and *Inter- ViewPoint* check actions. These are used to validate the consistency of a single ViewPoint or between multiple ViewPoints. The ViewPoint approach has been formalized using distributed Graph Transformation [18]. In our approach a common meta-model relates the different modelling notations that can be used, and the work plans are indeed graph transformation rules. The *In-* and *Inter-ViewPoint* check actions can be expressed as rules similar to the ones presented in section 5.4. The Pounamu tool [33] supports multiple views, which are related to a "glued" model via events (whose semantics are encoded in Java). This approach is quite similar to ours, but they don't consider abstract/concrete levels and is less formal as they do not use graph grammars, but the observer design pattern. Other very recent tool proposals [32] also consider structures similar to triple graphs in order to handle concrete and abstract syntax (e.g. the "bridge models" in [32]). The recent GMF (Graphical Modeling Framework) project [12] under Eclipse also considers different meta-models for concrete and abstract representations, together with an additional static model for mapping both syntaxes. This mapping also takes into account a tool model (which defines a buttons palette). In our approach, the mapping is richer, as in addition to a correspondence graph meta-model, the designer can specify triple rules to specify domain-specific behaviours. Moreover, we believe our work may serve as a theoretical foundation for other approaches.

Finally, note that although we have presented an example using the 1.5 version of the UML standard, these techniques are also applicable to UML 2.0, as similar problems can be found in many of the proposed diagrams.

8 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 IE-triple meta-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 meta-rules, which are equivalent to a number of concrete ones, where nodes and edges are replaced by each element in its inheritance clan. In this work, we have formalized triple graph grammars in the DPO approach, adapted the original work in [3] (regarding inheritance) to triple graphs, and extended it to allow edge refinement and application conditions (see Appendix A). These ideas are naturally applicable to the processing of VLs with multiple views.

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 diagram models with an existing abstract syntax model, generated by the previous definition of other diagrams. Event-driven rules have also been useful for layout management at the concrete syntax level. Besides event-driven grammars and the theoretical concepts, we have also presented other novel contributions from a practical point of view, such as an implementation of application conditions, inheritance concepts in rules, and triple rules in the AToM^3 environment.

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, and improve the automatic generation of environments for VLs with multiple views. The extension of the AToM^3 meta-model with spatial relations (in the style of [5]) is under consideration, also taking into account the OMG meta-model proposal for diagram definition and interchange [30].

Acknowledgements: This work has been partially sponsored by the Spanish Ministry of Education and Science with projects MOSAIC (TSI2005-08225-C07-06) and MODUWEB (TIN 2006-09678). The authors gratefully thank the referees for their useful and detailed suggestions.

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.* Sci. of Computer Programming 44, pp.: 181-203.
3. Bardohl, R., Ehrig, H., de Lara J., and Taentzer, G. 2004. *Integrating Meta Modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation.* Proc. ETAPS/FASE'04, LNCS 2984, pp.: 214-228. Springer.
4. Bottoni, P., Guerra, E., and de Lara, J. 2006. *Metamodel-based Definition of Interaction with Visual Environments.* Proc. of the MDDAUI'06, pp.: 43-46.
5. Bottoni, P., Costagliola, G. 2002. *On the Definition of Visual Languages and their Editors.* Proc. DIAGRAMS'02, LNAI 2317, pp.: 305-319. Springer.
6. 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.
7. Cordes, B., Hölscher, Kreowski, H-J. 2004. *UML Interaction Diagrams: Correct Translation of Sequence Diagrams into Collaboration Diagrams.* Proc. AGTIVE'03, LNCS 3062, pp.: 275-291. Springer.
8. Corradini, A., Montanari, U., Rossi, F. 1996. *Graph Processes.* Fundamenta Informaticae, vol. 6(3-4), pp.: 241-265. IOS Press.
9. de Lara, J., Ermel, C., Taentzer, G., Ehrig, K. 2004. *Parallel Graph Transformation for Model Simulation applied to Timed Transition Petri Nets.* Proc. GT-VMT'04, Electronic Notes in Theoretical Computer Science 109, pp.: 17-29. Elsevier.
10. de Lara, J., Vangheluwe, H. 2002. *AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling.* Proc. ETAPS/FASE'02, LNCS 2306, pp.: 174 - 188. Springer. See the AToM^3 page: `http://atom3.cs.mcgill.ca`
11. de Lara, J., Vangheluwe, H., Alfonseca, M. 2004. *Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3*. Software and System Modeling 3(3), pp.: 194-209.
12. Eclipse Graphical Modeling Framework (GMF) home page at: `http://www.eclipse.org/gmf/`
13. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. 1999. *Handbook of Graph Grammars and Computing by Graph Transformation.* (1). World Scientific.
14. Ehrig, H., Habel, A., Padberg, J., Prange, U. 2004. *Adhesive High-Level Replacement Categories and Systems* Proc. ICGT'04. LNCS 3256, pp.: 144-160. Springer.
15. Ehrig, H., Prange, U., Taentzer, G. 2004. *Fundamental Theory for Typed Attributed Graph Transformation* Proc. ICGT'04. LNCS 3256, pp.: 161-177. Springer.
16. Ehrig, H., Ehrig, K., Prange, U. and Taentzer, G. 2005. *Formal Integration of Inheritance with Typed Attributed Graph Transformation for Efficient VL Definition and Model Manipulation.* Proc. 2005 IEEE VL/HCC, pp.: 71-78. Dallas (USA).
17. Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., and Goedicke, M. 1992. *ViewPoints: A Framework for Integrating Multiple Perspectives in System Development*, Int. Journal of Software Engineering and Knowledge Engineering, vol. 2(1), pp.: 31-57.
18. Goedicke, M., Enders, B.E., Meyer, T., Taentzer, G. 1999. *Towards Integrating Multiple Perspectives by Distributed Graph Transformation* Proc. AGTIVE'99, LNCS 1999, pp. 369-377, Springer.

19. Grosse-Rhode, M., Bardohl, R., Simeoni. M. 2001. *Interactive Rule-based Specification with an Application to Visual Language Definition*, Proc. WADT'01, LNCS 2267, pp. 1-20, Springer.
20. Guerra, E., de Lara, J. 2003. *A Framework for the Verification of UML Models. Examples using Petri Nets.* Proc. JISBD'03. Alicante. Spain. pp.: 325-334.
21. Guerra, E., de Lara, J. 2004. *Event-Driven Grammars: Towards the Integration of Meta-Modelling and Graph Transformation.* Proc. ICGT'04, LNCS 3256, pp.: 54-69. Springer.
22. Guerra, E., Díaz, P., de Lara, J. 2005. *Supporting the Automatic Generation of Advanced Modelling Environments with Graph Transformation Rules.* Proc. JISBD'05. pp.: 67-74. Thomson.
23. Guerra, E., de Lara, J. 2006. *Attributed Typed Triple Graph Transformation with Inheritance in the Double Pushout Approach.* Technical Report UC3M-TR-CS-06-01 of the Universidad Carlos III (Madrid). Available at `http://www.ii.uam.es/~jlara/investigacion/techRep_UC3M.pdf`
24. Guerra, E., de Lara, J. 2006. *Model View Management with Triple Graph Transformation Systems.* Proc. ICGT'06, LNCS 4178, pp.: 351-366. Springer.
25. Heckel, R., Wagner, A. 1995. *Ensuring consistency of conditional graph rewriting - a constructive approach* Proc. SEGRAGRA 1995, ENTCS Vol 2, 1995.
26. Jansen, A.R, Marriott, K. and Meyer, B. 2003. *CIDER: A Component-Based Toolkit for Creating Smart Diagram Environments.* Proc. 9th Conference on Distributed and Multimedia Systems. pp.: 353-359.
27. Schürr, A. 1994. *Specification of Graph Translators with Triple Graph Grammars.* In LNCS 903, pp.: 151-163. Springer.
28. Taentzer, G., Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Varró, D., and Varró-Gyapay, S. 2005. *Model Transformation by Graph Transformation: A Comparative Study.* Model Transformation in Practice workshop at MODELS'05. Jamaica.
29. Taentzer, G., Rensink, A. 2005. *Ensuring Structural Constraints in Graph-Based Models with Type Inheritance.* Proc. of FASE'05, LNCS 3442, pp. 64-79.
30. *Unified Modeling Language: Diagram Interchange version 2.0* June 2005. Available at: `http://www.omg.org/docs/ptc/05-06-04.pdf`
31. UML specification at the OMG's home page: `http://www.omg.org/UML`.
32. Vargas, F., Roda, J. L., Estévez, A., Avila, O., Sánchez, E. V. 2006. *Generación de Editores Gráficos de Modelos para una Herramienta MDA*. Proc. DSDM'06 workshop at JISBD'06. Sitges (Spain). `http://www.dsic.upv.es/workshops/dsdm06`
33. Zhu, N., Grundy, J.C. and Hosking, J.G., 2004. *Pounamu: a meta-tool for multi-view visual language environment construction* Proc. IEEE VL/HCC, pp.: 254-256.

A Appendix: Theoretical Concepts of Triple Graph Transformation with Node and Edge Inheritance

This Appendix introduces the main theoretical concepts of triple graph transformation with node and edge inheritance in the DPO approach. For a full presentation, see [23].

A.1 Attributed Typed Triple Graphs

In order to define triple graphs, we start by using the concept of *E-graph* (extended graph) proposed in [15], which allows graphs to be attributed in their nodes and edges. Attribute values are stored in set V_D, and in addition to graph edges, two additional kind of edges are introduced to model attributes: node and edge attribution edges. The first ones allow nodes to have attributes, while the second ones model edge attributes.

Definition 1 *(E-graph) An E-graph is a tuple* $G = (V_G, V_D, E_G, E_{NA}, E_{EA}, (source_j, target_j)_{j \in \{G,NA,EA\}})$*, where* V_G *and* V_D *are sets of graph and data nodes respectively;* E_G *is a set of graph edges;* E_{NA} *and* E_{EA} *are sets of node and edge attribution edges; finally,* $source_j$ *and* $target_j$ *are functions defining the source and target of edges, defined as follows:*

- $source_G\colon E_G \to V_G$, $target_G\colon E_G \to V_G$.
- $source_{NA}\colon E_{NA} \to V_G$, $target_{NA}\colon E_{NA} \to V_D$.
- $source_{EA}\colon E_{EA} \to E_G$, $target_{EA}\colon E_{EA} \to V_D$.

Fig. 21 shows a diagrammatic representation of an E-graph, which depicts a sequence diagram (similar to the one in Fig. 1) containing two objects with an activation box each.

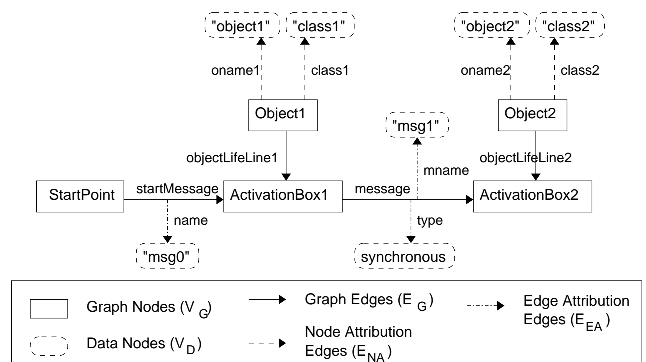

Fig. 21 An E-graph.

In addition to E-graphs, we also define mappings between two E-graphs. An E-graph morphism is a tuple of set morphisms, one for each set component in the E-graph (V_G, V_D, E_G, E_{NA}, E_{EA}). In addition, the structure of the E-graph should be preserved, that is, the $source_j$ and $target_j$ functions must commute with the morphisms.

Definition 2 *(E-graph morphism) Given two E-graphs* G^1 *and* G^2*, an E-graph morphism* $f : G^1 \rightarrow G^2$ *is a tuple* $(f_{V_G}, f_{V_D}, f_{E_G}, f_{E_{NA}}, f_{E_{EA}})$ *with* $f_{V_i} : V_i^1 \rightarrow V_i^2$ *and* $f_{E_j} : E_j^1 \rightarrow E_j^2$ *with* $i \in \{G, D\}$, $j \in \{G, NA, EA\}$*, where* f *commutes for all source and target functions.*

E-graphs together with E-graph morphisms form category **EGraph**. Next, we use E-graphs to build our notion of triple graphs (*TriE-graph*).

TriE-graphs are made of three E-graphs (source, correspondence and target) and two correspondence functions c_1 and c_2. The correspondence functions are defined from the nodes in the correspondence graph to a node or an edge in the other two graphs. In addition, the functions can be undefined, and this is modelled with a special element in the codomain (named "·"). Therefore, we have extended the previous notion of triple graphs [27] in several ways. First, we use a definition that contemplates attributes in nodes and edges. Second, our correspondence functions are more flexible, as the co-domain includes nodes and edges, and the special element for modelling that the function is undefined.

Definition 3 *(TriE-graph) A TriE-graph* $TriG = (G_1, G_2, G_C, c_1, c_2)$ *is made of three E-graphs* $G_i = (V_{G_i}, V_{D_i}, E_{G_i}, E_{NA_i}, E_{EA_i}, (source_{j_i}, target_{j_i})_{j \in \{G,NA,EA\}})$ *for* $i \in \{1, 2, C\}$*, with* $V_{D_1} = V_{D_2} = V_{D_C}$ *and two functions* $c_j : V_{G_C} \rightarrow V_{G_j} \cup E_{G_j} \cup \{\cdot\}$ *(for* $j = 1, 2$*).*

Graph G_1 is called source or concrete, graph G_2 is called target or abstract, and G_C is called correspondence. Functions c_1 and c_2 are called source and target correspondence functions respectively. We use the auxiliary sets $edges_i = \{x \in V_{G_C} | c_i(x) \in E_{G_i}\}$, $nodes_i = \{x \in V_{G_C} | c_i(x) \in V_{G_i}\}$ and $undef_i = \{x \in V_{G_C} | c_i(x) = \cdot\}$ for $i = 1, 2$. The latter set is used to denote that the correspondence function c_i for an element x is undefined. The previous two sets are used to denote that the codomain of the correspondence function c_i for an element x are edges or nodes, respectively. Morphisms c_1 and c_2 represent m-to-n relationships between nodes and edges in G_1 and G_2 via G_C in the following way: $x \in V_{G_1} \cup E_{G_1}$ is related to $y \in V_{G_2} \cup E_{G_2} \iff \exists z \in V_{G_C} \mid x = c_1(z)\, and\, y = c_2(z)$.

Fig. 22 shows a TriE-graph, which contains the abstract and concrete syntax of a UML sequence diagram. The target graph G_2 in the upper part corresponds to the abstract syntax, the source graph G_1 in the lower part corresponds to the concrete syntax, and the correspondence graph G_C in the middle contains elements relating both by means of the correspondence functions. Although the three E-graphs making a TriE-graph have the same data sets V_{D_i}, we have repeated the elements in each E-graph for clarity (i.e. element "class1" in V_{G_1} and V_{G_2} is the same). Moreover, we have only shown those data elements used for attribution.

Mappings between two TriE-graphs are made of three E-graph morphisms plus additional constraints regarding the preservation of the correspondence functions.

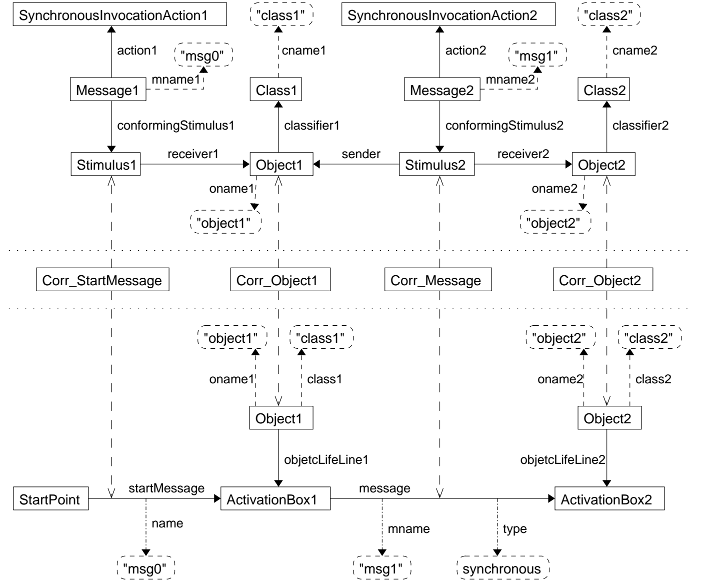

Fig. 22 TriE-graph with the Abstract and Concrete Syntax of a Sequence Diagram.

Definition 4 *(TriE-graph morphism) Given two TriE-graphs* $TriG^1$ *and* $TriG^2$*, a TriE-graph morphism* $f : TriG^1 \rightarrow TriG^2$ *is a tuple* $f = (f^1, f^2, f^c)$ *made of three E-graph morphisms* $f^i : G_i^1 \rightarrow G_i^2$ *(*$i \in \{1, 2, C\}$*) such that:*

- $f^i_{V_{G_i}} \circ c_i^1|_{nodes_i^1} = c_i^2 \circ f^C_{V_{G_C}}|_{nodes_i^1}$ *for* $i = 1, 2$ [5].
- $f^i_{E_{G_i}} \circ c_i^1|_{edges_i^1} = c_i^2 \circ f^C_{V_{G_C}}|_{edges_i^1}$ *for* $i = 1, 2$.
- $c_i^1|_{undef_i^1} = c_i^2 \circ f^C_{V_{G_C}}|_{undef_i^1}$ *for* $i = 1, 2$.

TriE-graphs and TriE-graph morphisms form category **TriEGraph** (see [23]), where the former are the objects, and the latter the arrows. It is indeed a category, as the identity arrow is the identity TriE-graph morphism, and the composition of TriE-graph morphisms is associative. Now, we provide TriE-graphs with an algebra over a suitable signature, in order to provide a structure to the data values (an organization into sorts) as well as operations.

Definition 5 *(Attributed Triple Graph) Given a data signature* $DSIG = (S_D, OP_D)$ *which contains sorts for attribution* $S'_D \subseteq S_D$*, an attributed triple graph* $TriAG = (TriG, D)$ *consists of a TriE-graph* $TriG = (G_1, G_2, G_C, c_1, c_2)$ *and one algebra* D *of the given DSIG signature with* $\biguplus_{s \in S'_D} D_s = V_{D_i}$ *for* $i \in \{1, 2, C\}$.

Mappings between two attributed triple graphs are made of a TriE-graph morphism and an algebra homomorphism. Again, attributed triple graphs together with attributed triple morphisms form the category **TriAGraph** (see [23]).

Now, we provide a typing to triple graphs by defining a triple type graph (similar to a meta-model triple). This

[5] $c_i^1|_A$ is the restriction of function c_i^1 of $TriG^i$ to the elements in set A.

is a special attributed triple graph, where the algebra is final. That is, the carrier set for each sort has a unique element, the sort name.

Definition 6 *(Attributed Type Triple Graph) An attributed type triple graph is an attributed triple graph* $TriATG = (TriTG, Z)$*, where* Z *is the final algebra of the* $DSIG$ *signature with carrier sets* $Z_s = \{s\}\ \forall s \in S_D$*.*

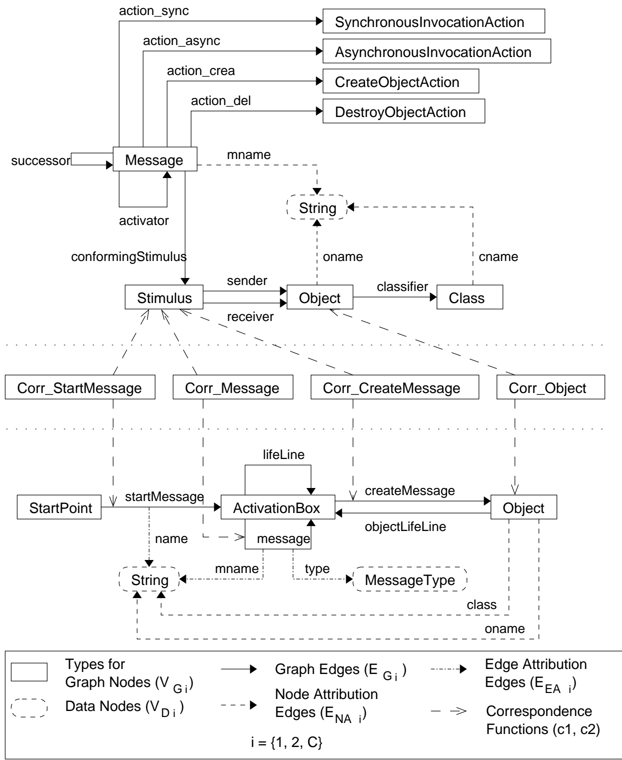

Fig. 23 Attributed Type Triple Graph for the Abstract and Concrete Syntax of Sequence Diagrams.

Fig. 23 shows an attributed type triple graph for the definition of both abstract and concrete syntax of UML sequence diagrams. The data signature is given by $DSIG = Char + String + MessageType$, where $Char$ is an auxiliary sort, and only $String$ and $MessageType$ are used for attribution. The target graph in the upper part of the triple graph corresponds to the abstract syntax (similar to the UML standard definition), the source graph in the lower part contains the concrete syntax, and the correspondence graph in the middle relates concepts of both sides. There are edge types in the concrete syntax (such as *startMessage*, *message* and *createMessage*) which are related to node types in the abstract syntax. Moreover, *ActivationBox*, *lifeLine* and *StartPoint* in the concrete syntax do not have an associated abstract syntax element. Finally, there are elements in the abstract syntax, such as *Message* (and the *successor* and *activator* edge types) and all the actions, which do not have an associated concrete element.

The typing of a triple graph is represented as a morphism from the graph to the type graph. That is, from now on, we work with objects that are tuples, storing information about the graph and the typing. This in fact can be formalized as a slice category.

Definition 7 *(Attributed Typed Triple Graph) An attributed typed triple graph (short ATT-graph) over* $TriATG$ *is an object* $TriTAG = (TriAG, t)$ *in the slice category* **TriAGraph/TriATG***, where* $TriAG = (TriG, D)$ *is an attributed triple graph and* $t\colon TriAG \to TriATG$ *is an attributed triple graph morphism called the typing of* $TriAG$*.*

Fig. 24 shows an ATT-graph over the attributed type triple graph in Fig. 23. In this figure, we use a UML-like notation, in which nodes and edges are labelled with their type (in the usual UML notation for instances), and their attributes are shown in a box. This is the notation that was used throughout the paper and that will be used in the remaining of the Appendix.

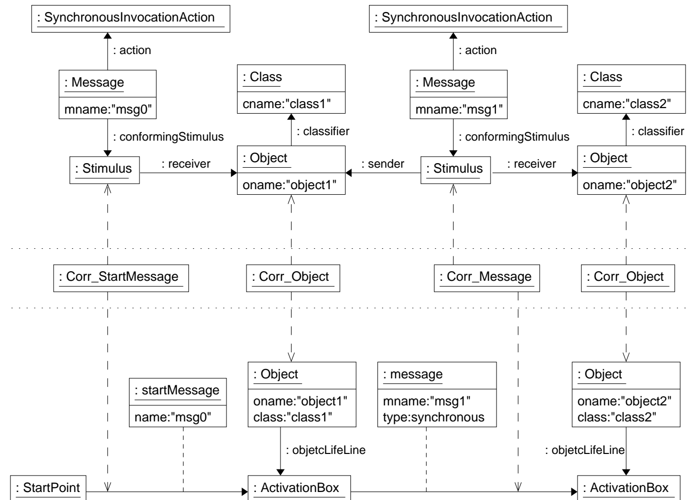

Fig. 24 Attributed Typed Triple Graph, with respect to the Attributed Type Triple Graph in Fig. 23.

Mappings between ATT-graphs (called ATT-morphisms) are like mappings between attributed triple graphs, but the morphism has to preserve the typing of the source triple graph. ATT-graphs over an attributed type triple graph $TriATG$, together with ATT-morphisms, form category $\mathbf{TriAGraph_{TriATG}}$ (see [23]).

Category **TriAGraph** is indeed isomorphic to a comma-category $\mathbf{ComCat(V_1, V_2; Id)}$, where V_1 and V_2 are forgetful functors. The first one goes from category **TriEGraph** to category **Set** and "forgets" the triple graph structure, taking just the set of data values of one of the graphs (as all the V_{D_i} sets are equal). The second functor V_2 goes from category $\mathbf{DSIG-Alg}$ to **Set**, placing together in a set the elements of the carrier sets for attribution (disjointly). The resulting comma-category

has objects $(TG, D, op: V_1(TG) \to V_2(D))$ which satisfy $V_1(TG) = V_2(D)$ and $op = id$. This category, and therefore $\mathbf{TriAGraph_{TriATG}}$, can be proved to be an adhesive HLR category [23]. This means that we can use the main results of graph transformation theory, as they have been lifted from graphs to adhesive HLR categories [14].

A.2 Attributed Typed Triple Graph Transformation

This section presents the main concepts and definitions of attributed typed triple graph transformation in the DPO approach. We start by defining the concept of triple rule.

Definition 8 *(Triple Rule) Given an attributed type triple graph $TriATG$ with data signature $DSIG$, a typed attributed triple graph rule (triple rule in short), $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ consists of three ATT-graphs L, K and R (typed over $TriATG$) with a common $DSIG$-algebra $T_{DSIG}(X)$ (which is the $DSIG$-termalgebra with variables X), and injective ATT-morphisms $l: K \to L$, and $r: K \to R$.*

In order to apply a triple rule p to an ATT-graph G (called *host* ATT-graph), an occurrence of the LHS should be found in the graph. That is, an ATT-morphism $m: L \to G$ needs to be found. Once the morphism is found, the rule is applied in two steps. In the first one, the elements in $m(L - l(K))$ are deleted from G, yielding graph D. In the second step, the elements from $R - r(K)$ are added to D, resulting in graph H. These two steps are modelled by two pushouts. A pushout is the gluing of two structures through some common elements. In **TriAGraph** and $\mathbf{TriAGraph_{TriATG}}$ pushouts are built componentwise, by calculating the pushout of each set in each one of the three E-graphs (see [23]).

Definition 9 *(Direct Derivation) Given a triple rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$, an ATT-graph $TriTAG$ and an ATT-morphism $m : L \to G$ (called match), a direct derivation $G \overset{p,m}{\Longrightarrow} H$ from G is given by the double pushout (DPO) diagram in category $\mathbf{TriAGraph_{TriATG}}$ shown in Fig. 25, where (1) and (2) are pushouts.*

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & (1) & \downarrow{\scriptstyle d} & (2) & \downarrow{\scriptstyle m^*} \\ G & \xleftarrow{l^*} & D & \xrightarrow{r^*} & H \end{array}$$

Fig. 25 Direct Derivation as DPO Construction.

Fig. 26 shows an example of direct derivation. The rule is typed over the attributed type triple graph shown in Fig. 23. It simply connects an object with its corresponding class, creating an edge (labelled "5" in R and H) in the abstract syntax model. For this purpose, the rule's LHS locates a class in the abstract syntax named as attribute "class" of the object in the concrete syntax.

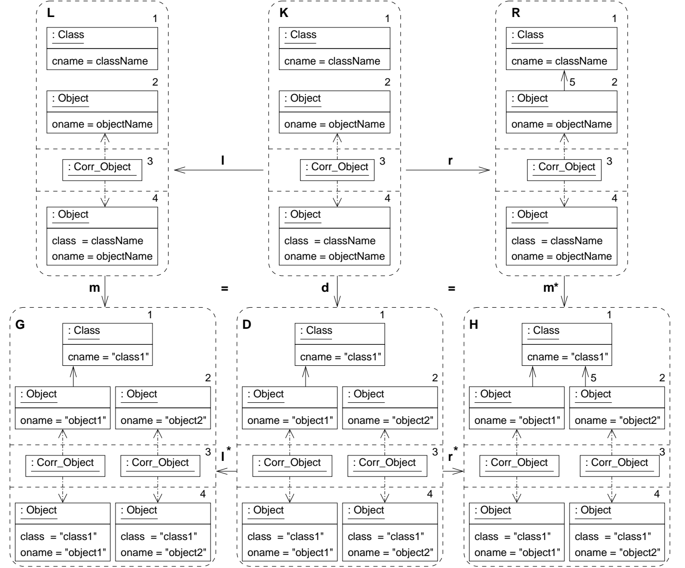

Fig. 26 A Direct Derivation Example.

Next, we define the concept of grammar and language.

Definition 10 *(Triple Graph Grammar and Language) A triple graph grammar $TGG = (DSIG, TriATG, P, TriAS)$ is made of a data signature $DSIG$, and attributed type triple graph $TriATG$, a set P of triple rules, and an initial ATT-graph $TriAS$, typed over $TriATG$. The language generated by TGG is given by $L(TGG) = \{TriTAG | TriAS \Rightarrow^* TriTAG\}$.*

In addition, we provide triple rules with application conditions, in the style of [25]. We first define conditional constraints on ATT-graphs. An application condition is then a conditional constraint on the L component of the triple rule.

Definition 11 *(Triple Conditional Constraint) A triple conditional constraint $cc = (x: L \to X, A)$ over an ATT-graph L consists of an ATT-morphism x and a set $A = \{y_j : X \to Y_j\}$ of ATT-morphisms. An ATT-morphism $m: L \to G$ satisfies a constraint cc over L, written $m \models_L cc$, iff $\forall n: X \to G$ with $n \circ x = m$ $\exists o: Y_j \to G$ (where $y_j : X \to Y_j \in A$) such that $o \circ y_j = n$ (see Fig. 27).*

$$\begin{array}{ccccc} Y_j & \xleftarrow{y_j} & X & \xleftarrow{x} & L \\ & {\scriptstyle o}\searrow & \downarrow{\scriptstyle n} & \swarrow{\scriptstyle m} & \\ & & G & & \end{array}$$

Fig. 27 A Triple Conditional Constraint Satisfied by m.

Roughly, the constraint is satisfied by morphism m if no occurrence of X is found in G, or if some is found, then an occurrence of some Y_j should also be found. If the set A is empty, then we have a negative application condition (NAC), where the existence of an ATT-morphism n implies $m \not\models_L cc$. Morphisms x and y_j are total, but we use a shortcut notation. In this way, the subgraph of L (resp. X) that does not have an image in X (resp. Y_j) is isomorphically copied into X (resp. Y_j) and appropriately linked with their elements.

We assign triple rules a set AC of triple conditional constraints (called application condition). For a rule to be applicable at a match m, it must satisfy all the application conditions in the set. Fig. 6 shows an example of two triple rules with NACs (the set A in the application condition is empty). Following the mentioned shortcut notation, in the NAC only the additional elements to the LHS and their context have been depicted.

A.3 Edge and Node Inheritance for Triple Graph Transformation

For the approach to be useful in meta-modelling environments, we extend attributed type triple graphs with inheritance relations. We use a similar approach to the one shown in [3] and [16], but we have adapted it to ATT-graphs, and extended it with edge inheritance. The extended type triple graphs with inheritance are defined like a normal type triple graph with two additional graphs for the node and edge inheritance hierarchies, and two sets of abstract nodes and edges. For technical reasons related to the inheritance of the correspondence function, multiple inheritance (for nodes) is forbidden in the correspondence graph. As in [29], we only allow an edge to inherit from another one, if the source and target nodes of the child edge belong to the children nodes of the source and target nodes of the parent edge. For this purpose, we use the notion of *clan* (see definition 13), which is a function that applied to a node or edge returns the set of all its children nodes or edges, including itself.

Definition 12 *(Attributed Type Triple Graph with Inheritance) An attributed triple type graph with inheritance (short meta-model triple)* $TriATGI = (TriATG, (VI_i, EI_i, AV_i, AE_i)_{i\in\{1,2,C\}})$*, consists of:*

- *An attributed type triple graph* $TriATG = (TriTG, Z)$.
- *Three node inheritance graphs*[6] $VI_i = (VI^i_V, VI^i_E, vs^i\colon VI^i_E \to VI^i_V, vt^i\colon VI^i_E \to VI^i_V)$ *with* $VI^i_V = V^i_G$*, for* $i \in \{1,2,C\}$*. Multiple inheritance is forbidden in the correspondence graph, therefore* $\forall n \in VI^C_V$, $|\{e \in VI^C_E | vs^C(e) = n\}| \leq 1$.
- *Three edge inheritance graphs* $EI_i = (EI^i_V, EI^i_E, es^i\colon EI^i_E \to EI^i_V, et^i\colon EI^i_E \to EI^i_V)$ *with* $EI^i_V = E^i_G$ *for* $i \in \{1,2,C\}$*. Moreover* $\forall e, e' \in EI^i_V, x \in EI^i_E$ *such that* $es^i(x) = e'$ *and* $et^i(x) = e$ *(i.e. e′ inherits from e), we have* $source_{G_i}(e') \in clan_{VI^i}(source_{G_i}(e))$ *and* $target_{G_i}(e') \in clan_{VI^i}(target_{G_i}(e))$.
- *Three sets* $AV_i \subseteq VI^i_V$*, for* $i = \{1,2,C\}$*, called abstract nodes.*
- *Three sets* $AE_i \subseteq EI^i_V$*, for* $i = \{1,2,C\}$*, called abstract edges.*

[6] A graph is made of a set of nodes (V), a set of edges (E) and source and target functions for the edges (s and t).

Fig. 4 shows an example meta-model triple, which is an extension of the attributed type triple graph in Fig. 23. We have collapsed each graph TG_i, node inheritance graph VI_i and edge inheritance graph EI_i in a unique graph. The edges of the inheritance graphs are shown with hollow edges (following the usual UML notation) and the elements in AV_i and AE_i are shown in italics. We treat "composition" edges (the ones with a black diamond) as any other edge.

Having meta-model triples, it is still possible to use the theory developed so far by "flattening" the attributed type triple graph with inheritance. This flattening operation makes explicit the semantic meaning to both kinds of inheritance (for nodes and edges) and leads to a normal attributed type triple graph. As usual, edges and attributes are inherited by subclasses, while only attributes are inherited by subedges. Thus, in the flattening operation, the inherited elements are explicitly copied down the inheritance hierarchy. In the current theory there is no support for attribute overriding, although in the correspondence graph we allow overriding of the correspondence functions. In this way, if a correspondence function is undefined for some node in the correspondence graph, then its value is obtained from the nearest node in the (node) inheritance path for which the function is defined.

But in order to use this approach, we do not want to use the flattened version of the type graphs. Instead, we have defined the typing directly from attributed triple graphs to meta-model triples (in a similar way as in [16]). These typing morphisms are no longer attributed triple morphisms, but a more general kind of morphism called triple clan morphism. These morphisms take into account the node and edge inheritance relations and correspond uniquely to the typing by the flattened type graph. We only define formally the inheritance clan concept; the interested reader can consult [23].

Definition 13 *(Node and Edge Inheritance clan) Given a meta-model triple* $TriATGI = (TriATG, (VI_i, EI_i, AV_i, AE_i)_{i\in\{1,2,C\}})$*, the node inheritance clan for each node* $n \in VI^i_V$*, is defined as* $clan_{VI^i}(n) = \{n' \in VI^i_V \mid \exists\ path\ n' \xrightarrow{*} n\ in\ VI_i\} \subseteq VI^i_V$ *with* $n \in clan_{VI^i}(n)$*. In a similar way, for each edge* $e \in EI^i_V$*, the edge inheritance clan is defined as* $clan_{EI^i}(e) = \{e' \in EI^i_V \mid \exists\ path\ e' \xrightarrow{*} e\ in\ EI_i\} \subseteq EI^i_V$ *with* $e \in clan_{EI^i}(e)$

For example, in Fig. 4, the node inheritance clan of node $ConcreteElement$ is $clan_{VI^1}(ConcreteElement) = \{ConcreteElement, ActivationBox, StartPoint, Object\}$.

The edge inheritance clan for $AbsMessage$ is $clan_{EI^1}(AbsMessage) = \{AbsMessage, createMessage, Message, StartMessage\}$.

Then, we can extend triple rules with the inheritance concept. We call these rules *inheritance-extended triple rules*, or IE-triple rules. In this way, nodes and edges in an IE-triple rule can be typed by node and edge types (also called classes and associations) in the meta-model triple, which may be refined by a number of sub-classes and sub-associations. As mentioned in section 3.2, an IE-triple rule typed in that way is equivalent to a set of *concrete* IE-triple rules, resulting by the valid substitutions of each node and edge in the IE-triple rule by all the concretely typed nodes and edges in its inheritance clan. If the set of equivalent rules of an IE-triple rule has cardinality greater than one, the IE-triple rule is called *IE-triple meta-rule*. We first define type refinement and then define IE-triple rules.

Definition 14 *(Type Refinement) Given attributed triple graph* $TriAG = (TriG, D)$ *with* $TriG = (G_1, G_2, G_C, c_1, c_2)$ *and* $G_i = (V^G_{G_i}, V^G_{D_i}, E^G_{G_i}, E^G_{NA_i}, E^G_{EA_i}, (source^G_{j_i}, target^G_{j_i})_{j\in\{G,NA,EA\}})$ *for* $i \in \{1, 2, C\}$*, and two clan morphisms* $type\colon TriG \rightarrow TriATGI$ *and* $type'\colon TriG \rightarrow TriATGI$*,* $type'$ *is called a* type refinement *of type, written* $type' \leq type$[7] *if:*

- $type'^i_{V_G}(n) \in clan_{VI^i}(type^i_{V_G}(n))$, $\forall n \in V^G_{G_i}$, *for* $i \in \{1, 2, C\}$.
- $type'^i_{E_G}(n) \in clan_{EI^i}(type^i_{E_G}(n))$, $\forall n \in E^G_{G_i}$, *for* $i \in \{1, 2, C\}$.
- $type'^i_X = type^i_X$, *for* $X \in \{V_D, E_{NA}, E_{EA}\}, i \in \{1, 2, C\}$.
- $type'_D = type_D$.

Definition 15 *(Inheritance-Extended Triple Rule) An inheritance-extended triple rule (short IE-triple rule), is a triple rule typed by a meta-model triple* $TriATGI = (TriATG, (VI_i, EI_i, AV_i, AE_i)_{i\in\{1,2,C\}})$ *and is given by* $p = (L \xleftarrow{l} K \xrightarrow{r} R, type, AC)$*. The first element is an attributed triple graph rule (l and r are attributed triple morphisms);* $type = (type_i\colon i \rightarrow TriATGI)_{i\in\{L,K,R\}}$ *is a triple of typing triple clan morphisms, one for each part of the triple rule;* $AC = \{cc_i = (x_i\colon L \rightarrow X_i, type_{X_i}, A_i = \{(y_{ij}\colon X_i \rightarrow Y_{ij}, type_{Y_{ij}})\})\}$ *is a set of application conditions where* $type_{X_i}\colon X_i \rightarrow TriATGI$ *and* $type_{Y_{ij}}\colon Y_{ij} \rightarrow TriATGI$ *are triple clan morphisms, such that the following conditions hold:*

- $type_L \circ l = type_K = type_R \circ r$ *(the type of the preserved elements is the same in L, K and R).*
- $type^i_{R,V_G}(V'^R_{G_i}) \cap AV_i = \emptyset$, *where* $V'^R_{G_i} := V^R_{G_i} - r^i_{V_G}(V^K_{G_i})$, *for* $i \in \{1, 2, C\}$ *(no new node in R is abstractly typed).*
- $type^i_{R,E_G}(E'^R_{G_i}) \cap AE_i = \emptyset$, *where* $E'^R_{G_i} := E^R_{G_i} - r^i_{E_G}(E^K_{G_i})$, *for* $i \in \{1, 2, C\}$ *(no new edge in R is abstractly typed).*
- $type_{Y_{ij}} \circ y_{ij} \leq type_{X_i}$ *and* $type_{X_i} \circ x_i \leq type_L$ *for all* $cc_i \in AC$ *(the typing of* Y_{ij} *is finer than the typing of* X_i*, and this is finer than the typing of L).*
- $type^i_{L,V_G} \circ c^L_i \circ l^C_{V^C_G}|_{nodes^K_i} = type^i_{K,V_G} \circ c^K_i|_{nodes^K_i} = type^i_{R,V_G} \circ c^R_i \circ r^C_{V^C_G}|_{nodes^K_i}$ *for* $i = 1, 2$ *where* c^K_i*,* c^L_i *and* c^R_i *are the correspondence functions of K, L and R (the typing of the target nodes of the correspondence functions in K are the same in L and R).*
- $type^i_{L,E_G} \circ c^L_i \circ l^C_{V^C_G}|_{edges^K_i} = type^i_{K,E_G} \circ c^K_i|_{edges^K_i} = type^i_{R,E_G} \circ c^R_i \circ r^C_{V^C_G}|_{edges^K_i}$ *for* $i = 1, 2$ *(the typing of the target edges of the correspondence functions in K are the same in L and R).*
- *The datatype part of L, K, R,* X_i *and* Y_{ij} *is* $T_{DSIG}(X)$*, the term algebra of DSIG with variables X, and* $l_D, r_D, x_{i_D}, y_{ij_D}$ *are identities (data preserving).*

The top row of Fig. 28 shows a simple IE-triple meta-rule example (a detailed version of the one shown in Fig. 7). The rule identifies the activator message of another one, creating an edge in the abstract graph. Nodes 7, 8 and 9 and edges 10 and 11 of the concrete graph have an abstract typing. The meta-rule is equivalent to four concrete rules. Node 7 can take types *StartPoint* or *ActivationBox* in the concrete rule, node 8 has to be an *ActivationBox*, and node 9 can be an *Object* or an *ActivationBox*. Thus, four combinations are possible, where the edge types are determined by the choice of node types (see the comments for the simplified rule in section 3.2).

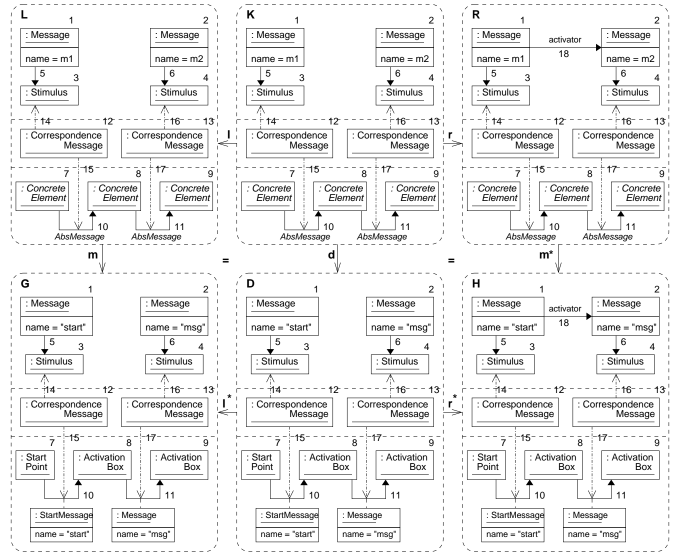

Fig. 28 An Example of IE-Triple Meta-Rule and Derivation.

In order to apply an IE-triple meta-rule to a triple graph, a structural match with respect to the untyped rule has to be found. The typing of the match should be concrete and finer than the type of the rule's LHS. Moreover, the typing of the target of the correspondence functions in the host graph should also be finer than in

[7] we say that *type'* is finer than *type*

the rule's LHS. Finally, the match should satisfy the application conditions. The direct derivation can be built by first constructing the double pushout in **TriAGraph**, yielding the attributed triple graph H. Then, the typing is added. The preserved elements by the rule do not change their type. The new elements take their type from R, as the elements added by the rule should have a concrete typing. Fig. 28 shows a direct derivation example, where abstract elements 7, 8, 9, 10 and 11 in the rule take concrete types *StartPoint*, *ActivationBox*, *ActivationBox*, *StartMessage* and *Message* in graph G.

The application of a meta-rule is equivalent to the application of one of its concrete rules. Moreover, it is possible to show that the language generated by an IE-extended triple graph grammar is the same as the language generated by a triple graph grammar without inheritance. The latter grammar uses as the type graph the flattening of the meta-model triple, and as rules the concrete rules of each meta-rule in the former grammar (see [23] for the details).

B Appendix: Formal Definitions for Event-Driven Grammars

This appendix shows the precise definitions of event-driven graph grammar and derivation that was used in section 4.

Definition 16 *(Event-Driven Graph Grammar) An event-driven graph grammar* $edGG = (DSIG, TriATGI, evt, pre, sys{-}act, post, del, M_i)$ *is made of a data signature* $DSIG$*, a meta-model triple* $TriATGI$*, five sets of IE-triple rules and an initial attributed triple graph* M_i *typed by* $TriATGI$*. The following constraint holds:* $\forall p = (L \xleftarrow{l} K \xrightarrow{r} R, type, AC) \in evt \cup sys - act \cup del$, $X_j^L = X_j^K = X_j^R = \emptyset$*, for* $X = \{V_G, E_G, E_{NA}, E_{EA}\}$*,* $j \in \{2, C\}$*. That is, the abstract and correspondence graphs of rules in* evt*,* $sys - act$ *and* del *are empty.*

Definition 17 *(Event-Driven Graph Grammar Derivation) Given an event-driven graph grammar* $edGG = (DSIG, TriATGI, evt, pre, sys{-}act, post, del, M_i)$*, a direct event-driven graph grammar derivation starting from* M_i *is depicted as* $M_i \Longrightarrow\!\!\!\!\Rightarrow M_f$ *, and consists in the composition of the attributed typed triple derivations shown in Fig. 8. An event-driven graph grammar derivation is depicted as* $M_i \overset{*}{\Longrightarrow\!\!\!\!\Rightarrow} M_f$ *and consists of zero or more direct event-driven graph grammar derivations.*