Metamodel-based definition of interaction with visual environments[*]

Paolo Bottoni
University "La Sapienza"
Dep. Computer Science
Rome, Italy
bottoni@di.uniroma1.it

Esther Guerra
Universidad Carlos III
Dep. Computer Science
Madrid, Spain
eguerra@inf.uc3m.es

Juan de Lara
Universidad Autónoma
Dep. Computer Science
Madrid, Spain
jdelara@uam.es

ABSTRACT

Metamodel approaches to building visual environments are becoming common in the field of domain specific visual languages, mainly focusing on the definition of visual editors and of simulation environments. Recent efforts tackle the generation of complex interaction management both in the editing and in the executing phases. We present an approach to interaction specification which takes into account metamodel information both on the objects that can be manipulated and on the spatial relations among them. Interaction dynamics are defined through a visual, declarative and formal notation based on graph grammars.

Categories and Subject Descriptors

Software [**SOFTWARE ENGINEERING**]: Design Tools and Techniques—*User Interfaces*

General Terms

Design, Human Factors, Theory

1. INTRODUCTION

Metamodelling frameworks for diagrammatic languages definition and management are particularly exploited to construct environments for Domain Specific Visual Languages (DSVLs), as they allow a rapid implementation of visual environments based on some abstract notion of visual entities and of relations among them [3, 4]. These environments must support interaction to create visual sentences in the defined language and to manipulate the entities in them. The different interaction techniques should be constrained to allow only transformations complying with the metamodel. However, current approaches rely on standard forms of interaction, typically recurring to mechanisms provided by the implementation language, with little or no formal reference to the metamodel. This imposes rigidities on the possible interactions, or forces to explicitly programming the different alternatives for activating the same transformation.

We propose here the integration of a metamodel for interaction – I – with one for diagrammatic languages – D – thus decoupling the definition of low-level user-generated *events* from that of abstract high-level *visual actions*, to be served with reference to the constraints embodied in D. To this end, we rely on previous work separately developed by the authors aimed at defining metamodels for diagrammatic languages syntax and semantics [3] and integrating some level of formality in the management of user interactions [5]. In particular, we exploit the notion of *family* of diagrammatic languages [2, 3] and put it to work in combination with event-driven grammars [5], which were originally not related to the management of spatial relations. This kind of graph grammars supports a stratified view of interaction events where patterns of user-generated events can be mapped to more refined high-level ones, which can in turn produce cascading effects. This allows a complete configurability of the user interface, so that different styles of interaction can be used to produce the same effect, or a same user-generated event can produce different effects, according to the interface modality. Relations between low- and high-level events can be dynamically modified by substituting a grammar with another, without having to reprogram the event listeners.

Paper organization. After presenting related work on formal definition of user interaction in Section 2, we discuss the integration of the I and D metamodels in Section 3. Section 4 presents event-driven graph grammars and discusses their use to manage different levels of abstraction in event definition. Finally, Section 5 presents an application to the management of the containment relation between nested states in a simple variant of Statecharts, while Section 6 draws conclusions and points to directions for future work.

2. RELATED WORK

A formal model of interactive tasks and components must lie at the basis of every proposal for their integration and management. We do not consider here task-related formalisms, such as ConcurTaskTrees [6], and concentrate on models of components and abstract interaction.

Models of interaction control are either *centralized*, with some high-level machine driving legal interaction, (e.g. [9]), or *distributed*, by associating with every interactive component its own control mechanism. A formal approach to the definition of centralized interaction control is proposed in [1], leading to the definition of verifiable finite state systems from a Visual Event Grammar. An important effort in the direction of distributed control is the proposal of Interactive Cooperative Objects (ICOs) [8], which encapsulate state and behaviours, modelled through Petri nets, of Virtual Reality objects with reference to the events that can have effect

[*]Work supported in part by the EC's Human Potential Programme under contract HPRN-CT-2002-00275, SegraVis, and the Spanish Ministry of Science and Education, projects MD2 (TIC200303654) and MOSAIC (TSI2005-08225-C07-06).

on them and the way in which they react to these events (business logic processes and rendering algorithms). ICOs abstract from the specific devices through which events are produced, by mapping generated events to services managing them.

The Abstract User Interaction (AUI) approach to graphical user interfaces [11] maps concrete user interactions, depending on different devices and style choices, to abstract ones. Realisations of concrete interaction techniques for an abstract interaction are provided on request, in a lazy functional style. The consequences of an interaction are modelled through calls to external functions. AUI is mainly aimed at device-independence, but still attaches computations to low-level events. In a similar way, [10] proposes an abstract definition of interface components as composition of platform independent widgets and views, to be mapped to their concrete realisation on a platform dependent model.

While we capitalize on the distinction between concrete and abstract events, we allow for a wider scope of supported interactions, as we rely on a metamodel including the base classes for the visual elements in a DSVL, for the spatial relationships between these elements, and for the GUI elements with which the end user can interact, as well as classes for the different kinds of events and actions. Having an explicit representation of actions resembles the concept of "action languages", which are becoming popular to express the semantics of metamodel-based languages [7]. However, our approach adds flexibility in that modelling the semantics of events by means of graph grammar rules allows the expression of complex conditions on their management as subgraphs in the left-hand side of a rule.

3. METAMODEL INTEGRATION

In a diagrammatic language, significant *spatial relations* exist among *identifiable elements*. The latter are recognizable entities in the language, to which a semantic role can be associated, each element being univocally materialized by means of a *complex graphic element*. Each such element is composed in turn of simpler *graphic elements*, each possessing one or more *attach zones*, which define its availability to participate in different spatial relations, such as containment or touching. The existence of a spatial relation with semantic relevance between two elements is assessed via the predicate `isAttached()` implemented by each realisation of `AttachZone`. Figure 1 shows the metamodel D including these concepts. For space constraints, an abridged version is presented (for a complete presentation, see [2, 3]). In particular, we restrict our analysis to *direct* spatial relations, which are here always regarded as binary, without considering emergent relations, such as those derived from closure of direct ones. A visual environment also contains GUI elements, e.g. buttons. Typically, an automatically generated environment would contain a button for creating each element of the DSVL alphabet Σ, as well as buttons for performing visual actions (e.g. selecting, moving or deleting).

In order to relate interaction concepts with those in D, we introduce a notion of *interaction support*. Low-level events generated on an interaction support are thus related to the corresponding visual element. Their effects can extend to other elements, for example by navigating spatial relations.

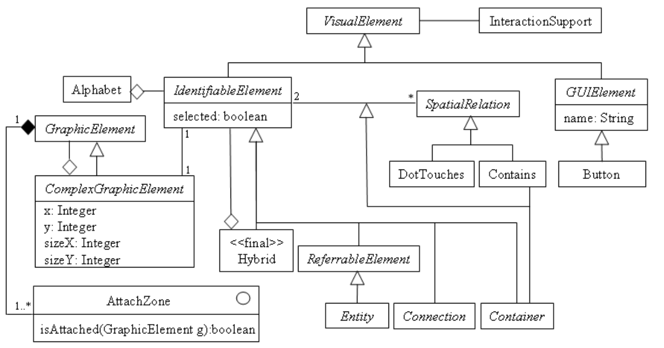

Figure 1: Metamodel for Diagrammatic Languages.

Figure 2 shows the metamodel I for interaction. Low-level events are received by some interaction support. Let Σ be the set of `IdentifiableElement` concrete subclasses. A typical set of low-level events for Σ is $Low = \{click < X, x, y, time > [mod](\sigma), drag < X, x, y, time > [mod](\sigma), drop < X, x, y, time > [mod](\sigma)\}$, where $[mod]$ indicates some combination of key modifiers and X, x, y, and $time$ indicate a mouse button, a screen position and a time respectively. Moreover, general scope events can occur on the canvas on which visual entities are depicted or on the GUI elements. The actual set of events depends on the characteristics of the underlying event support system.

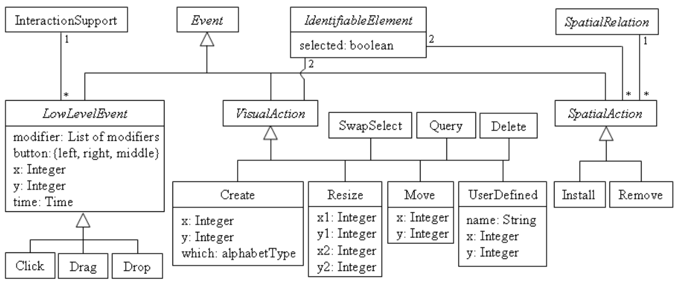

Figure 2: Metamodel for Interaction.

At a higher level, a set *Act* of *visual actions* defines the types of significant interactive actions a user can perform, again related to the identifiable elements to be manipulated. Typically, *Act* is such that $\forall \sigma \in \Sigma$, $Act \supset \{create(\sigma), swapSelect(\sigma), delete(\sigma), move(\sigma), resize(\sigma), query(\sigma)\}$, having omitted action-specific parameters. *Act* can be recursively enriched by letting designers define new types of actions, based on events in *Low* and *Act*.

We introduce a new type of high-level events for spatial relations, such that a (direct) spatial relation can be brought to bear, or cease to exist, between two identifiable elements, as an effect of such events, constituting a set *RelAct*. Let Θ be the set of subtypes of `SpatialRelation`. We have that $\forall \theta \in \Theta, \forall \sigma_1, \sigma_2 \in \Sigma$ such that instances of θ relate pairs of type (σ_1, σ_2), $RelAct \supset \{install(\theta, \sigma_1, \sigma_2), remove(\theta, \sigma_1, \sigma_2)\}$. We adopt here event-driven graph grammars both to describe the mapping of user-generated events (in *Low*) to high-level events (in $Act \cup RelAct$) and to specify their effects.

4. EVENT-DRIVEN GRAPH GRAMMARS

Event-driven graph grammars were proposed in [5] as a means to handle user interaction in the editing phase. They make the events that the visual elements can receive explicit, and model the actions to be done upon event generation via rules. An event-driven grammar is made of *pre-rules* and *post-rules* to be applied before and after the event is actually executed, and complements the DSVL metamodel with interaction dynamics.

Event management occurs in five steps. First, the user generates an event on an interaction support, which is attached to the associated element. Then, pre-rules are executed as long as possible. They can modify model elements and produce and delete events. Hence, they can be thought of as pre- conditions (failing which the event is deleted and not executed) and pre-actions for a given event. In the third step, the existing event(s) are actually executed. At this point, zero or more events may be associated to various model elements. In the fourth step, the post-rules are executed as long as possible. Finally, the events are deleted.

Event-driven rules may contain instances of abstract classes in their left and right hand sides (LHS and RHS respectively). Although no instance of abstract classes can be present in the model, "abstract objects" in rules can be matched to any concrete subclass instance. This feature makes rules more compact and reusable. For example, rules specifying the behaviour of *Containers* will be valid for any language containing entities that inherit from this class.

Figure 3 shows a pre-rule that transforms a low-level event into a visual, high-level action, modelling entity movement in a *drag and drop* interaction modality. We use a compact notation for the rules, in which LHS and RHS are presented together. The elements to be added by the rule application are shown in bold, and those to be removed in dashed lines. For rules with negative application conditions (NACs), the elements that should not be present are shown in a gray area. Finally, if an attribute of an entity is modified, its value appears as a tuple containing the values before and after the rule application.

Figure 3 shows a pre-rule checking whether some identifiable element has received a *Drop* low-level event in its interaction support, while the "select" button in the user interface is selected. In this case, it generates a *Move* high-level visual action associated with the element. This rule uses "abstract objects" and therefore is valid for any DSVL environment.

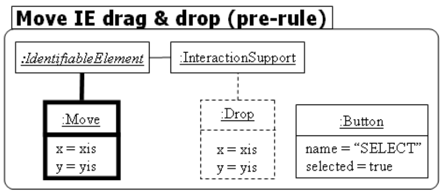

Figure 3: Movement in Drag and Drop Modality.

Event-driven rules can model other interaction modalities (like point and click or grasp and stretch) rewriting patterns of low-level events into high-level actions. Figure 4 shows a set of rules modelling a point and click behaviour for moving entities,. The user has already entered the *MOVE* modality by selecting the corresponding button, which provokes the deselection of any selected identifiable element in the canvas. The first rule selects the clicked element and deletes the click event. The second rule looks for a *click* event which is not associated to the interaction support area of any element (i.e. the click was on the canvas), and then generates a *Move* visual action associated with the selected element.

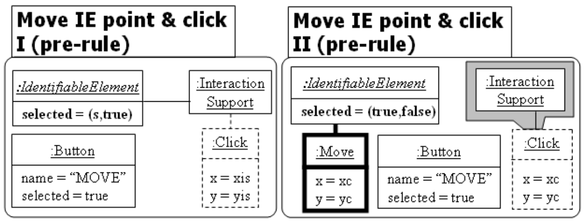

Figure 4: Movement in Point and Click Modality.

When generating an environment, the DSVL designer can choose between these interaction modalities. In addition, the high-level actions can be interpreted in different ways by different sets of rules, providing an additional degree of customization, as shown in the next Section.

5. AN EXAMPLE: MANAGING CONTAINMENT IN STATECHARTS

In this Section we show how to customize containment handling in an environment for a simplified version of Hierarchical State Machines, defined by the metamodel of Figure 5. The visual entities refine the relevant classes of containment- and connection-based families of languages [3]. In particular, a *State* may act both as a container for other states, and as source or target of a connection (the *Transition*). Class *Substate* is introduced as a specialization of *Contains*, whose instances relate pairs of instances of *State*.

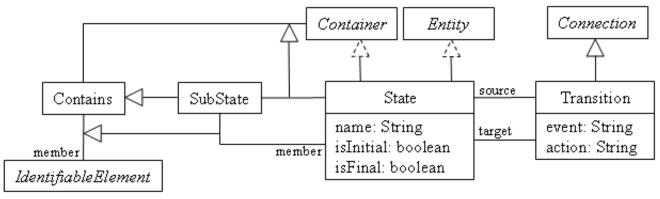

Figure 5: Metamodel for Statecharts.

The DSVL designer can configure specific behaviours for handling the different spatial relations (containment, alignment, adjacency, etc.) We present an example where we model different behaviours that may occur when one element is moved outside a container. In the first one, the containee is disconnected from the container. In the second, the container is resized to accommodate the new position of the containee. In the third one, we forbid a containee to be moved outside the container. The main idea is that the fact of moving a containee outside a container will produce the user-defined event *"Take Out"*. Hence, we make available three rules interpreting the event in three different ways.

Figure 6 shows a pre-rule that generates the take-out user-defined event when an element is moved outside its container. As this is a pre-rule, the *Move* action has not been performed yet. This rule has a NAC that forbids applying the rule more than once in a row.

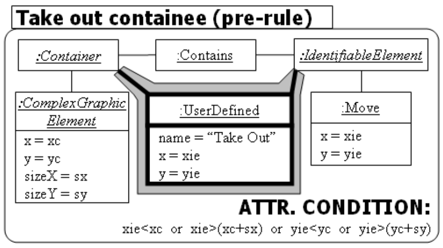

Figure 6: Generation of a User-Defined Action.

Figure 7 shows three different interpretations for the take-out event. The *Detach containee* rule removes the event and adds a *Remove* spatial action. The *Resize container* rule replaces the event by a resize attached to the container, with the appropriate coordinates for resizing. Finally, rule *Not allowed* deletes both the take-out and the *Move* event of the containee, thus preventing its movement.

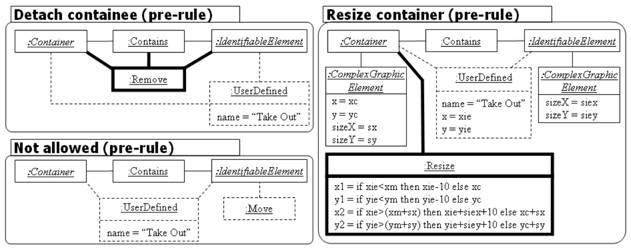

Figure 7: Three Interpretations of "Take Out".

A movement of a state that concludes by placing it within a different container leads to the installation of a new spatial relation between the moved state and the destination one. In this case a new user-defined event, named "Place In", would be generated in a pre-rule analogous to Figure 6.

6. CONCLUSIONS AND FUTURE WORK

We have presented a novel approach to the definition of interaction modalities for DSVL environments based on a metamodel. Low-level events and high-level actions are taken into account, together with spatial relations and DSVL concepts. Different behaviours can be customized by means of rules. The definition of complex interaction patterns is supported by separating the processing of events from a management policy for all their envisaged consequences.

The advantages of such an approach are manifold. First, using graph grammars enables a formal approach to interaction definition which is also visual and declarative, thus favoring reuse of rules and reasoning on them. Moreover, as rules are defined at an abstract level, they are independent of the concrete user interface. This decoupling of device-originated events from low-level abstract events, and from the management of high-level actions, facilitates the definition of configurable and adaptive environments. Event-driven graph grammars provide a visual, formal semantics to an action language for DSVL environments. The integration of different GUI elements (such as buttons) into D and their inclusion in the elements manageable through event-driven grammars is a further step towards a complete formal definition of the user interactions.

The approach is being integrated into the ATOM^3 architecture. Future work will study other interaction phenomena. For example, the issue of grouping elements and defining actions which simultaneously affect all of them can be tackled by viewing all the elements as contained in a temporary dummy container. Other challenges include global phenomena, such as context switch or complex layout redefinition.

7. REFERENCES

[1] J. Berstel, S. Crespi-Reghizzi, G. Roussel, and P. San Pietro. A scalable formal method for design and automatic checking of user interfaces. *ACM Transaction on Software Engineering and Methodology*, 14(2):124–167, 2005.

[2] P. Bottoni and G. Costagliola. On the definition of visual languages and their editors. In *Diagrams*, pages 305–319, 2002.

[3] P. Bottoni and A. Grau. A Suite of Metamodels as a Basis for a Classification of Visual Languages. In *Proc. of 2004 IEEE VL/HCC*, pages 83–90. IEEE CS, 2004.

[4] J. de Lara and H. Vangheluwe. AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling. In *Proc. of FASE'2002*, volume 2306 of *LNCS*, pages 174–188. Springer, 2002.

[5] E. Guerra and J. de Lara. Event-Driven Grammars: Towards the Integration of Meta-Modelling and Graph Transformation. In *Proc. of ICGT'2004*, volume 3256 of *LNCS*, pages 54–69. Springer, 2004.

[6] G. Mori, F. Paternò, and C. Santoro. CTTE: Support for Developing and Analyzing Task Models for Interactive System Design. *IEEE Trans. Software Eng.*, 28(8):797–813, 2002.

[7] P. Muller, P. Studer, F. Fondement, and J. Bezivin. Platform independent web application modeling and development with netsilon. *Software and System Modeling*, 4(4):424–442, 2005.

[8] D. Navarre, P. A. Palanque, R. Bastide, A. Schyn, M. Winckler, L. P. Nedel, and C. M. D. S. Freitas. A formal description of multimodal interaction techniques for immersive virtual reality applications. In *INTERACT*, volume 3585 of *LNCS*, pages 170–183. Springer, 2005.

[9] G. D. Penna, B. Intrigila, and S. Orefice. An environment for the design and implementation of visual applications. *J. Vis. Lang. Comput.*, 15(6):439–461, 2004.

[10] T. Schattkowsky and M. Lohmann. Towards Employing UML Model Mappings for Platform Independent User Interface Design. In *MDDAUI*, volume 159 of *CEUR Workshop Procs.*, 2005.

[11] K. A. Schneider and J. R. Cordy. Abstract user interfaces: A model and notation to support plasticity in interactive systems. In C. Johnson, editor, *DSV-IS*, volume 2220 of *LNCS*, pages 28–48. Springer, 2001.