Visual Specification of Multi-View Visual Environments

J.C. Grundy†, W.B. Mugridge†† and J.G. Hosking††

†Department of Computer Science
University of Waikato
Private Bag 3105, Hamilton, New Zealand
email: jgrundy@cs.waikato.ac.nz

††Department of Computer Science
University of Auckland
Private Bag 92019, Auckland, New Zealand
email: {rick,john}@cs.auckland.ac.nz

Abstract

We describe a set of visual tools for specifying and generating multi-view visual environments. JComposer provides an architecture description language for defining environment repositories, view models, and view-repository mappings. A visual event-flow language permits annotation of JComposer diagrams with event handlers specifying environment semantics. BuildByWire supports constraint-based visual specification of graphical elements for JComposer-based environments. JVisualise provides support for visualisation, querying and end-user modification of environment semantics.

1. Introduction

Advanced software development environments are large, complex pieces of software. They provide multiple tools to manipulate an evolving software artefact represented in a variety of notations. For example, an integrated CASE tool may provide editors for object-oriented analysis notations, such as those of UML [12], entity-relationship diagrams, user interface specification notations, and textual programming language code. Facilities are needed for managing consistency and inconsistency between representations, and between multiple views of a single representation [13, 8]. A common approach is to have a common repository fully describing software artefacts, with mappings to and from tool based representations and views which maintain consistency between the various models (or at least indicate where things are inconsistent) [8, 28]. Often these mappings are implemented using event propagation approaches [27, 26, 33].

There has been much work on simplifying the construction of such environments by abstracting common techniques into reusable frameworks. Attention has also focussed on the generation of environments and editing tools from meta-model descriptions, which are typically expressed in a visual notation [25, 27, 7]. To fully support such an approach, tool meta models need to support more than just a structural definition of the environment; they must also be capable of specifying dynamic behaviour. As a minimum the following aspects need to be specified:

- Shared repository objects and their inter-relationships
- Tool views of the repository and the mappings between the views and repository.
- Inter-object event propagation and response.
- The visual form and behaviour of view objects, ie the visual syntax and semantics of the tools

We describe a collection of visual languages to support these specification needs, with implementations that support the generation of environments directly from these visual specifications.

2. Our approach

Our approach uses change propagation and response graphs (CPRGs) as the underlying implementation architecture [28]. CPRGs propagate changes, such as the modification of an icon, as *change description* objects between *component* objects via *relationship* objects. Receiving objects interpret or store change descriptions appropriately to maintain consistency. CPRGs support the implementation of a wide range of tool capabilities, including multiple views, generic undo/redo, view versioning, and collaborative editing.

To specify CPRG components and relationships, we have developed the JComposer architecture description language. This language has a visual form containing elements in common with entity relationship diagrams and UML class diagrams. JComposer specifications are used to specify both the repository and view level components of an environment, together with the structural relationships between them, and the events (change descriptions) propagated along the relationships.

The semantics of event handling are specified using a visual filter-action language, abstracted from our earlier work on software process modelling [30], with event handlers specified using "event-flow" programming.

The visual form and editing semantics of graphical views and their components are specified using BuildByWire, a constraint-based notation editor. Textual editors, such as for programming code, are not currently supported, but can be conventionally implemented and integrated via an event processing interface.

In the following sections we describe these notations and illustrate the process in which they are used to specify and implement a multiple view UML use case diagram editor. Fig 1 shows the tool (UMLTool) in use. Use case diagrams contain actors (people shapes) and use cases (ellipses). Actors make use of use cases (arrow) and use cases may use or generalise other use cases.

Multiple use case diagrams can be constructed, with shared information (actors, etc). This UMLTool example is simple, as it only requires only a single notation with simple mappings to the common repository, but serves to illustrate our visual specification notations. We have specified and generated other multi-view, multi-user environments, including an ER modeller, a process modelling environment, a visualisation tool, and our specification tools themselves [29, 31].

2. Specification of the repository

Figure 2 shows part of the repository meta-model specification for UMLTool using JComposer. CPRG component types are represented as rectangles, and relationship types as ovals. Component-relationship links have associated arities. In this example, a Repository component has two one-to-many relationships, Actors and Use Cases, with, respectively, BaseActor and BaseUseCase components. BaseActors have a BaseParticipation relationship with BaseUseCases, which in turn have an optional BaseGeneralisation relationship to other BaseUseCases.

Component and relationship types specialise (ie inherit from) CPRG framework component and relationship types. For example, the Actors and UseCases relationships are both specialisations of the MVHashTableRel hash table relationship. The supertype name is shown directly under the type name in the icon. Component properties can be specified on the diagrams e.g., BaseUseCases has a UseCaseName property.

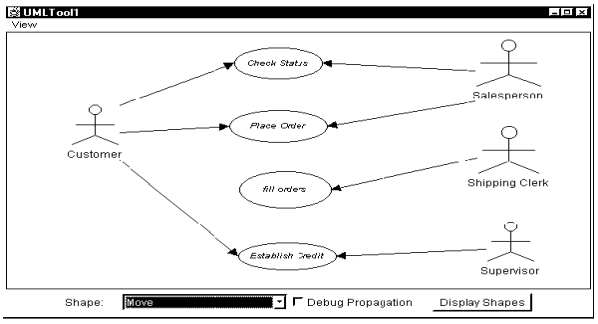

Figure 1. A UMLTool view

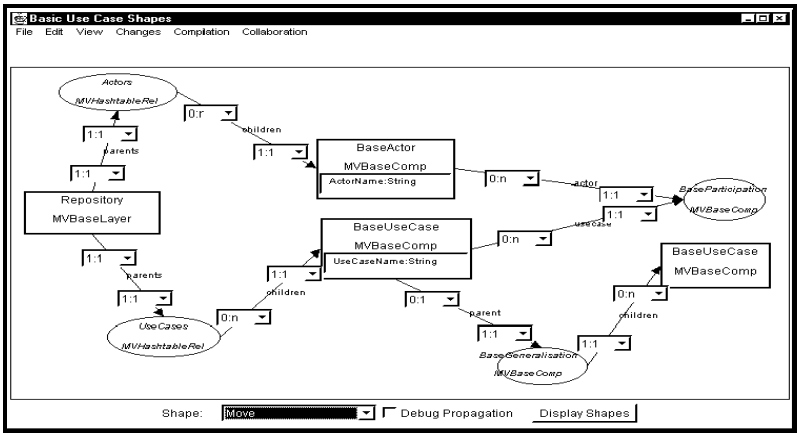

Figure 2. Part of the UMLTool repository specification

3. Specification of event handling

In the CPRG model, change descriptions (events) are generated whenever components or relationships are modified (eg properties changed). These event representations are then propagated to associated relationships or components for them to react to. The reactions, which may, for example, modify the receiving component's state, are used to implement semantic constraints on the environment.

Some component types have predefined actions that are performed on receipt of specific types of event. For example, an MVHashtableRel component listens to the components it indexes, and when any of the attributes that produce the hashtable key changes, it generates a ChangeKey event.

In general, however, application specific actions must be defined to specify the semantics of a tool. JComposer provides a notation to support event handler specification, using a visual event-flow programming style. *Filters* (rectangular icons) and *actions* (shaded ovals) can be attached, by event flows (arrowed lines), to components and relationships in order to detect and react to selected events.

Filters allow only selected (matched) events to flow through them. An action carries out some process, based on the event. Actions may, for example, automatically make further changes, undo an invalid change, store information, or inform users of a potential problem. They may also generate new events flowing out of them.

For example, the top of Figure 3 shows another view of the repository structure of Figure 2, but with an action (CheckUniqueValues) attached to the UseCases relationship. This action is passed all events received by the relationship. It aborts any insertion or key change if a duplicate key will result.

Filters and actions can be packaged into parameterisable "routines" for reuse. For example, the CheckUniqueValues filter/action model has been packaged in this way, and the lower part of Figure 3 shows its internal filter-action implementation. Two filters (rectangular icons) detect the addition of a BaseUseCase (EstablishRel(BaseUseCase) event) or modification of a BaseUseCase name (PropertyChange(UseCaseName)), via event flow connections (arrowed lines) from the UseCases relationship. The IsUniqueKey filter determines if the new/changed use case name is non-unique, i.e. another local use case has the same name. If so, it informs the user of this semantic error using the NotifyUserOfError action (shaded icon) and then "aborts" (reverses) this invalid editing operation using the AbortOperation action. The event input/output and component usage by this filter/action model are specified by input (e.g. "rel changed") and output (e.g. "change OK") items parameterising the detailed specification of the model.

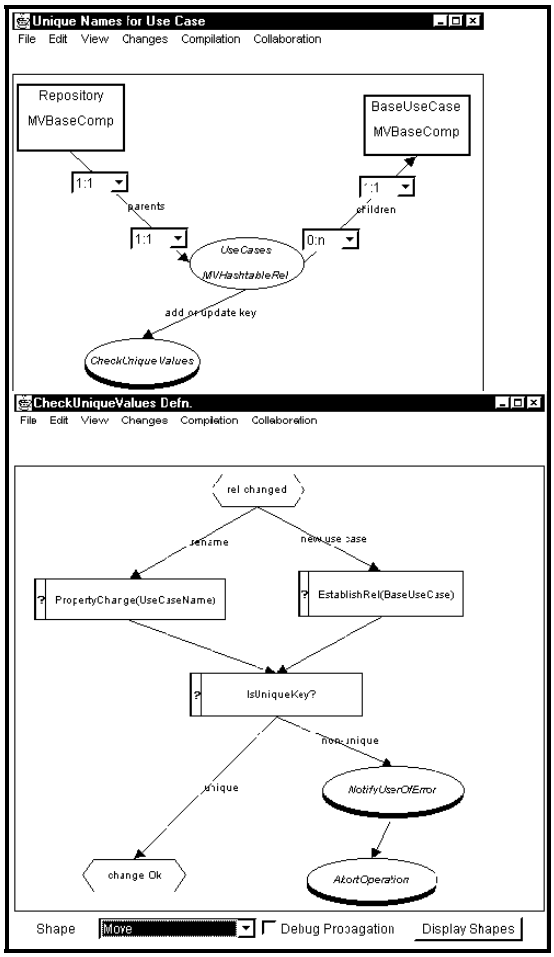

Figure 3: Example filter and action handler

A range of simple filters and actions are available for reuse in JComposer. Programmers can also package reusable filter and action models as in Figure 3, and implement new ones in Java. Event flows in JComposer filters and action models are similar in nature to the flow of data between processes in dataflow languages, but differ in the way event generation is used to drive execution rather than data synthesis via calculation.

4. Specification of view models

Atool designer may specify the details of each view of an underlying repository meta model. A view provides a partial representation of the repository, independent of the graphical form that icons in the view may take.

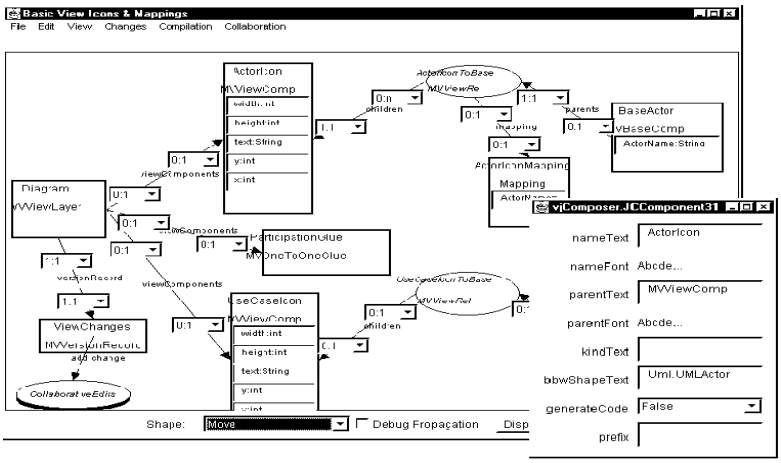

Figure 4. View model and view mapping specification.

Each structural component of a view is connected by a *view relationship* to a corresponding repository component. Associated "mapping" components specify the mapping of view component changes to the repository component and vice-versa.

Figure 4 shows a partial specification of a Use Case diagram view for UMLTool. The view itself is represented by the *Diagram* component, which has one-to-many relationships with components defining use case icons (*ActorIcon*, *UseCaseIcon*) and the connector icon used to connect them (*ParticipationGlue*). Also shown are view-repository mapping relationships, *ActorIconToBase* and *UseCaseIconToBase*. The former's mapping component, *ActorIconMapping*, specifies how view-level attributes map to/from repository attributes.

Event handlers can be attached to view relationships to map more complex structural changes to and from views and the repository. For example, a filter and action specification can ensure that whenever a new actor is attached to a use case in the repository, the graphical views of the use case have an appropriate new actor icon added and linked to the use case icon in the view.

Other kinds of views, for example a textual representation of a use case, can have mappings specified. When the base use case is modified, change description events generated can be unparsed into a human-readable textual form and inserted into the view's text, informing users of changes affecting the text.

Event handlers can also be attached to view level components and relationships. For example, an additional event handler is shown attached to the Diagram (view) component in Fig. 4 to support collaborative editing with other view users by broadcasting change descriptions between users' environments.

5. Specification of visual form

The visual form of view level components and their layout and editing semantics are specified using BuildByWire (BBW). BBW allows a visual notation designer to:

- compose the graphical elements (icons) that make up a visual notation, such as actors, use cases, and participation connectors;
- specify how those icons may be created, changed, connected, and transformed (such as through resizing);
- specify interface tools to manage their manipulation.

New icons are constructed by "wiring" together existing icons using constraints and containers. Icons available for composing include simple shapes like boxes, ovals, and lines, together with any (lightweight) Java user interface component, such as a TextField, Menu, ComboBox or Button. Icons have associated properties that may be changed at design time through property sheets (eg size, position, filling). They also have "reshaping" handles that are used for transforming (resizing, rotating, etc).

Other icons are containers that organise collections of icons, such as in vertical lists, organisational chart structures, card stacks and other layouts. In addition, layout may be specified with constraints.

Wires are multi-way constraints on the properties of figures, such as for alignment. They are added by dragging between handles, but don't usually have a visible iconic form of their own (a hierarchical

component view allows them to be edited). Some also introduce their own handles, which may be used to attach further wires. Using wires, the relative position of component figures can be specified, as well as the way in which repositioning and reshaping (resizing, rotating, swivelling, shearing) of one component affects another.

Figure 5 shows an example of the Actor icon being defined in BuildByWire for UMLTool. It is constructed from an oval, several lines, and some text. *Proportional* wires break the icon into four equal sections vertically. A proportional wire connects between two existing handles, with a new handle created between them. Whenever either of the original handles are moved (or resized), the new handle maintains its proportional position between them. The new handle can be "adjusted" to be at any proportional position between the two handles. The oval is placed in the top section of the icon, with *equality* wires joining the top and bottom of the oval to the top and bottom of the upper section. The oval is constrained to be circular by specifying that its width is equal to its height.

The arm and leg endpoints of the Actor are defined as offsets from the left- and right-hand handles of the icon using *relative* wires. These create a new handle at a fixed offset from a target handle. Whenever the target handle moves, the relative wire's handle moves correspondingly.

The text is placed at the bottom of the icon and its contents specified using a property sheet for the TextShape (shown in the bottom-right of Figure 5).

A property sheet (top right in Figure 5) is also used to specify which properties of a composed shape are accessible to underlying CPRG components. In this case, the only property of interest is the text. This single exported property is also displayed in the bottom-left window of Fig. 5, where the developer has also entered the class name to be associated with the icon.

Once auxiliary handles are hidden (in this case they are only needed during construction), the class corresponding to this new icon can be generated. BuildByWire generates a JavaBean class for each new composite icon of the notation.

Connectors, such as the participation connector of UMLTool are specified in a similar manner to icons. However, connectors are not freestanding, but are dependent on other icons to attach their endpoints to.

The next step is to specify the tool interface provided to the user to create and manipulate icons in a particular view. This involves selecting the required tool types and locations (eg pop up menu or tool bar) using a property sheet approach. BuildByWire generates user interface code (a JavaBean class) which includes the tools, and an interface to CPRG-based code for programmatic creation and manipulation of views, icons and connectors.

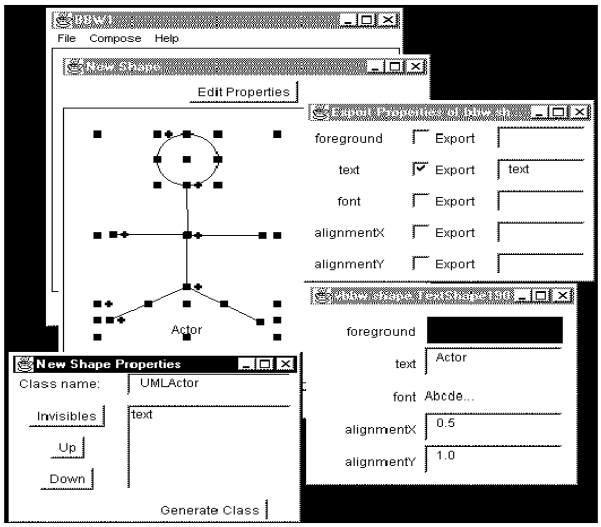

Figure 5. Specification of an Actor in BBW

JComposer view components are associated with appropriate BBW icons by specification of the icon type in the property sheet of the view component (as shown in Figure 4 right, where the UMLActor icon is specified as the icon for the ActorIcon component). As BBW icons are Java Beans, JComposer automatically extracts the public property names from the specified icon class and the user of JComposer specifies which of these should be represented in the JComposer view component. When these JComposer view component properties change, the generated JComposer view component updates the corresponding BBW icon property. Similarly, when the icon properties change, the JComposer view component detects this and updates its own propeties appropriately. JComposer generates this mapping code automatically. JComposer can also modify characteristics of BBW shapes programmatically to implement, for example, automatic expansion and elision facilities.

6. Environment generation

Once a tool has been modelled in JComposer, the tool developer can request generation of Java Beans implementations of the tool components, relationships, filters, actions, and icons. Unmodified or reused components are not regenerated; instead, instantiations of the reusable Java Bean implementation of the component are created when the tool is used.

Backend code, such as code generators, is added by tool implementors as textual code in subclasses of the JComposer-generated classes. Regeneration of an

environment will not modify these subclasses, which will only require hand amendment if a generated class change in a way that affects the subclass code.

7. Visualisation of generated tools

Tools developed using JComposer can be "statically" extended by modifying the JComposer definition. However, one advantage of our component-based modeling and implementation of tools is that the tools can be dynamically extended by adding or modifying components when the tool is in use. The JVisualise tool provides repository querying and user-defined event handling facilities in generated JComposer environments, using a similar notation to JComposer. Ids and type names replace type and supertype names respectively in the icons. Links between components are shown as relationship icons, and attribute values are shown instead of types. JVisualise views are updated as the environment executes, and relationship links can be highlighted as events are propagated along them.

Visualisations are constructed either interactively, by the user selecting running components to visualise via pop-up menus, or via a visual query language. Figure 6 shows a user-constructed visualisation of a running JComposer view, showing iconic visualisations of the component which represents the view (JComp. JCDiagram) and a component representing a JComposer component specification (JComp.JCompIcon). These "existing" components were added to the view by the user first requesting the view be visualised via a menu option. The user then asked for the viewComponents link for the view to be shown, and selected one JComp.JCCompIcon view component linked to the view to be shown. The user then requested nameText, x and y attributes of the view component to also be shown.

The user has also interactively added filter, action and version record (which stores component changes) components to the view in Figure 6. Together these cause all changes made to the diagram component (the view) to be stored. This can be used, for example, to support asynchronous cooperative work by making the changes available for perusal by other users. Unlike most existing tools, our interactively extensible JVisualise views allow users to readily extend their environment's event handling behaviour.

Users can also "query" an environment for components of interest. This is done by interactively constructing a visual query, using the same JVisualise notation, but with icons not linked to existing components. Constraints such as attribute values and link arities can be specified, and the query run. Components matching the query are visualised using the query structure itself [31].

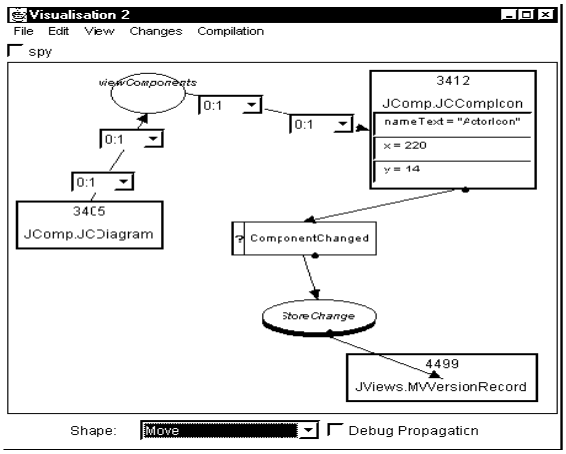

Figure 6. Example visualisation in JVisualise

8. Related work

A wide range of tools have been developed to facilitate the specification and generation of multiple-view systems. These range from the design and implementation of visual languages, to meta-CASE tools for specifying CASE tools, to the development of general-purpose multiple view environments.

Examples of visual language and environment specification tools include RGG [1], Escalante [7], Clockworks [27], Vampire [6], and DV-Centro [22]. Several of these, including Vampire and RGG use interpreted specifications and thus provide environments fully-generated from visual notations. Such generated environments tend to be rather limited in the degree of user interaction they provide, however, and have limited tool integration and multiple view consistency management support.

Several tools, such as Escalante and Clockworks, adopt partial-generation approaches, where some of a new environment is specified visually or textually in the generator tool, the rest being hand-implemented. Clockworks provides a similar component-based approach to JComposer for specifying and generating component structures, including limited support for mulitiple view specifications. It does not, however, provide visual event specifications but require tool developers to code these in the Clock language, using uni-directional constraints to keep views consistent. We have found such constraint approaches to lack sufficiently flexible view consistency management for our tool problem domains [28]. Escalante uses a range of visual and textual views to specify and then generate

visual language tools. Its visual views are limited to component structure specification and icon representation, lacking event processing specifications.

Meta-CASE tools are used to specify and generate CASE tools, many of which include visual language aspects. Examples include KOGGE [25] and MetaEDIT+ [3]. Tools for general-purpose software development using visual specifications include JBuilder [18], and Visual Javascript [11]. Most metaCASE tools provide repository and view specifications similar to JComposer's, but use textual constraints rather than visual event specification models. For example, KOGGE uses a JComposer-like notation for meta-model specification, but complex textual constraints rather than visual event handling for behaviour specification and multiple view consistency. Similarly, metaEDIT+ uses database rules to specify simple repository constraints.

JComposer filters and actions provide significant advantages over other approaches such as constraints and attribute grammars [15, 19], selective broadcasting [13], action routines [19], and the basic component event handling models of systems such as Visual Age [2] and JBuilder [18]. This includes the ability to readily understand and extend visual event handling models, an easily composed visual notation vs. textual formulae or code, and the ability to package filter and action models for reuse via a single visual item [32].

Many tools supporting the generation of editors for visual representations use textual specification of icon appearance and editor behaviour. Examples include Garnet [9], Amulet [10], Zeus [21], Unidraw [17], KOGGE [25], and MetaEDIT+ [3]. These toolkit-based approaches lead to complex specifications which bare little resemblance to the actual icons and editing behaviour desired. In particular, the specification of icon layout and editing constraint in these systems is complex and often unintuitive. BBW utilises a compositional technique with designers building accurate representations of editor components, with layout constraints and editing behaviour specified using a similar metaphor.

Other visual approaches to specifying visual language editors include those of VisiTile [4], Vampire [6], AgentSheets [14], DV-Centro [22], HotDoc [23], and Escalante [7]. The majority of these approaches use iconic composition approaches like BBW, but provide a limited range of fixed interaction capabilities (such as detecting mouse clicks). Specifying icons which can both dynamically resize or be resized by users, connectors that dynamically resize and reposition annotations, and a wide range of user interaction facilities are generally not supported, with the notable exception of DV-Centro. The compositional metaphor of BBW also allows designers to extend its built-in iconic, connector and constraint components for reuse.

Dynamic visualisation and repository querying tools include those of Teorey et al [16], Bird [20], Consens and Mendelzon [24], and metaEDIT+ [5]. Most use either graphical query languages, with results presented textually, or textual query languages with textual or graphical results. We have found the JVisualise approach of graphical structure representation and textual constraints, plus visualisation of results using the graphical structures from the query, to be highly suitable for component-based tools. The ability of tool users to easily add extra event processing behaviour to running tools is not supported visually in most other systems. Our JVisualise visual notation has an advantage of ease of use for novice users, but also expressive power for experienced tool developers.

7. Summary

We have described an approach to specification and generation of multi-view visual environments based on the JComposer, BuildByWire, and JVisualise tools. This provides an integrated approach to visual specification of components, their visual forms and possible user interactions. The approach to repository and view mapping specification is novel in its integration of static meta modelling and dynamic event handling, both specified visually. Reuse of this notation for visualising and debugging the executing environment is also novel.

One important consequence of our work is that we have lowered the cost of constructing editors and environments for "small" visual notations, and experimenting with and tailoring those notations quickly to need. Our component based approach also provides the ability to rapidly compose these "small" notations together either at the view level or at the repository level to provide multi-notation environments. Thus, rather than a "one notation does all" approach, JComposer tends to encourage a confederation of small, relatively orthogonal visual notations. An example of this is JComposer itself, which integrates several small notations (the architecture description language, the filter action language, and the BuildByWire compositional tool) in a powerful synergy.

Acknowledgements

The first author acknowledges funding from the New Zealand Foundation for Research, Science and Technology. The second and third authors acknowledge research grants from the University of Auckland Research Committee.

References

[1] Zhang. D.Q. and Zhang, K., "Reserved graph grammar: a specification tool for diagrammatic VPLs," in *Proceedings of the 1997 IEEE Symposium on Visual Languages,* IEEE CS Press, 1997, pp. 284-291.

[2] *IBM Visual Age for Java™*, IBM, http://www.software.ibm.com/ad/vajava, 1997.

[3] Kelly, S., Lyytinen, K., and Rossi, M., "Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment," in *Proceedings of CAiSE'96,* Lecture Notes in Computer Science 1080, Springer-Verlag, Heraklion, Crete, Greece, May 1996, pp. 1-21.

[4] Lewicki, D. and Fisher, G., "VisiTile - A Visual Language Development Toolkit," in *Proceedings 1996 IEEE Symposium on Visual Languages,* IEEE CS Press, Boulder, September 1996, pp. 114-121.

[5] Liu, H., "A Visual Interface for Querying a CASE Repository," in *Proceedings of the 1995 IEEE Symposium on Visual Languages,* IEEE CS Press, 1995, pp. 21-28.

[6] McIntyre, D.W., *Design and implementation with Vampire*, Visual Object-Oriented Programming. Manning Publications, Greenwich, CT, USA, 1995, chap. 7, pp. 129-160.

[7] McWhirter, J.D. and Nutt, G.J., "Escalante: An Environment for the Rapid Construction of Visual Language Applications," in *Proceedings of the 1994 IEEE Symposium on Visual Languages,* IEEE CS Press, 1994.

[8] Meyers, S., "Difficulties in Integrating Multiview Editing Environments," *IEEE Software*, vol. 8, no. 1, 49-57, January 1991.

[9] Myers, B.A., "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces," *COMPUTER*, vol. 23, no. 11, 71-85, 1990.

[10] Myers, B.A., "The Amulet Environment: New Models for Effective User Interface Software Development," *IEEE Transactions on Software Engineering*, vol. 23, no. 6, 347-365, June 1997.

[11] *Netscape Visual JavaScript™*, Netscape`Inc., http://www.netscape.com/compprod/products/tools/visual_js.html, 1998.

[12] Rational, C., *UML Document Set Version 1.1*, Rational`Corporation, http://www.rational.com/uml/references/, 1997.

[13] Reiss, S.P., "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software*, vol. 7, no. 7, 57-66, July 1990.

[14] Repenning, A., "Bending the Rules: Steps toward Semantically enriched Graphical Rewrite Rules," in *Proceedings of IEEE Symposium on Visual Languages,* IEEE CS Press, Darmstadt, Germany, September 1995, pp. 226-233.

[15] Reps, T. and Teitelbaum, T., "Language Processing in Program Editors," *COMPUTER*, vol. 20, no. 11, 29-40, November 1987.

[16] Teorey, T.J., Yang, D., and Fry, J.P., "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model," *Computing Surveys*, vol. 18, no. 2, 197-222, June 1986.

[17] Vlissides, J.M. and Linton, M., "Unidraw: A framework for building domain-specific graphical editors," in *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology,* ACM Press, 1989, pp. 158-167.

[18] *Borland JBuilder™*, http://www.borland.com/jbuilder.

[19] Backlund, B., Hagsand, O., and Pherson, B., " Generation of Visual Language-oriented Design Environments," *Journal of Visual Languages and Computing* , vol. 1, no. 4, 333-354, 1990.

[20] Bird, B., "An Open Systems SEE Query Language," in *Proceedings of 7th Conference on Software Engineering Environments,* IEEE CS Press, Netherlands, April 5-7 1995.

[21] Brown, M.H., " Zeus: A System for Algorithm Animation and Multi-View Editing," in *Proceedings of the 1991 IEEE Symposium on Visual Languages,* IEEE Computer Society Press, 1991, pp. 4-9.

[22] Brown, P.C., "Satisfying the graphical requirements of visual languages in the DV-Centro framework," in *Proceedings of the 1997 IEEE Symposium on Visual Languages,* IEEE CS Press, 1997, pp. 84-91.

[23] Buchner, J., Fehnl, T., and Kuntsmann, T., "HotDoc a flexible framework for spatial composition," in *Proceedings of the 1997 IEEE Symposium on Visual Languages,* IEEE CS Press, 1997, pp. 92-99.

[24] Consens, M., Medelzon, A., and Ryman, A., "Visualising and querying software structures," in *Proceedings of the 14th International Conference on Software Engineering,* IEEE CS Press, Melbourne, Australia, May 1992.

[25] Ebert, J., Suttenbach, R., and Uhe, I., "Meta-CASE in practice: A Case for KOGGE," in *Proceedings of the 9th International Conference on Advanced Information Systems Engineering,* LNCS 1250, Springer-Verlag, Barcelona, Spain, 1997, pp. 203-216.

[26] Gamma, E., Helm, R., Johnston, R., and Vlissides, J., *Design Patterns*. Addison-Wesley, 1994.

[27] Graham, T.C.N., Morton, C.A., and Urnes, T., "ClockWorks: Visual Programming of Component-Based Software Architecture," *Journal of Visual Languages and Computing*, 175-19, July 1996.

[28] Grundy, J.C., Hosking, J.G., and Mugridge, W.B., "Supporting flexible consistency management via discrete change description propagation," *Software - Practice & Experience*, vol. 26, no. 9, 1053-1083, September 1996.

[29] Grundy, J.C., Mugridge, W.B., and Hosking, J.G., "A Java-based toolkit for the construction of multi-view editing systems," in *Proceedings of the Second Component Users Conference,* Munich, Germany, July 14-18 1997.

[30] Grundy, J.C. and Hosking, J.G., "Serendipity: integrated environment support for process modelling, enactment and work coordination," *Automated Software Engineering*, vol. 5, no. 1, .

[31] Grundy, J.C., Hosking, J.G., and Mugridge, W.B., "Static and Dynamic Visualisation of Software Architectures for Component-based Systems," in *Proceedings of SEKE'98,* IEEE CS Press, San Francisco, June 18-20 1998.

[32] Grundy, J.C., Hosking, J.G., and Mugridge, W.B., "Support for end user specification of workflows, work coordination and tool integration," *Journal of End User Computing*, vol. 10, no. 2, .

[33] Hill, R.D., " The Abstraction-Link-View Paradigm: Using Constraints To Connect User Interfaces to Applications," in *Proceedings of CHI '92: Human Factors in Computing,* ACM Press, 1992, pp. 335-342.