Vixels, CreateThroughs, DragThroughs and Attachment Regions in BuildByWire

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

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

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

Abstract

BuildByWire is a direct-manipulation meta-editor for composing sophisticated visual notations and their editors from JavaBean components. It in turn generates JavaBeans, which can be plugged into other tools. BuildByWire has been used to generate editors for a variety of visual languages and notations. We describe new features that eliminate several previous limitations of BuildByWire and its generated editors. The designer of a notation now has more control over the tools that are provided in the generated editor. Composition of notational elements using layout managers has been improved. Connectors between notational elements have been made more general and flexible.

Key words: user-interface, visual notation, meta-editor, JavaBeans

1. Introduction

Visual notations are important in many domains, including software development, architectural design and business process reengineering. Many software applications require editors for visual notations that provide appropriate editing mechanisms. Examples include software design tools supporting notations such as the Unified Modelling Language (UML) [7], process modelling tools supporting work process diagrams [8], and building design tools supporting Computer-Aided Design (CAD) diagramming [17].

Developing editors for visual notations has proven to be a time-consuming and difficult task [1, 11, 15]. Not only do visual notational constructs have to be designed, but also layout and semantic constraints, editing functions and editing tools which support these notations. Many attempts at general-purpose meta-tools for the development of graphical editors have been produced. Examples include Unidraw [17], Zeus [1], Escalante [11], and BuildByWire [12]. However, many of these systems suffer from complex specification of tools using programming languages, inflexible constructs and notational symbols, inappropriate editing mechanisms in the resultant tools, and lack of reuse of existing visual notational elements.

We have developed the BuildByWire meta-editor which adopts a compositional approach to notation and editor specification and generation. Editor designers compose basic notational elements, or even third-party complex elements, along with layout and connectivity constraints and editing facilities. Particularly novel aspects of BuildByWire includes its support for reuse of JavaBeans components as visual elements, or *vixels*, *create throughs* used to support iconic element creation, *drag throughs* used to support iconic element layout, and *attachment regions* used to support iconic component interconnection. These facilities greatly improve the usablity of both the BuildByWire meta-tool and the editing tools it generates.

In the next section we illustrate the uses of BuildByWire, and identify limitations with previous versions of the tool. We then describe the process used to develop BuildByWire-based editing tools, with generic JavaBeans-based editors being produced. In the following sections, we look at various extensions to BuildByWire that improve usability of the tool, including create throughs, drag throughs and attachment regions. We then overview related research and finish with our current and future work.

2. Problem Domain

BuildByWire is a direct-manipulation meta-editor, a tool for composing sophisticated visual notations and generating editors for them. Originally developed in Prolog [12], BuildByWire has been redeveloped in Java to utilise and generate JavaBeans [9]. JavaBean components (such as panels, text fields and buttons) are used to compose the visual elements (*vixels*) of a notation, such as the vixel for a class of a UML class diagram.

Once the vixels of a notation have been composed and their beans generated, BuildByWire generates a visual editor bean. This provides the end-user with a direct-manipulation interface to create and modify diagrams or visual programs in that notation.

We have used BuildByWire to develop visual languages for several CASE tools [6, 7, 8, 12]. Such visual language editors support the end-user in creating diagrams from vixels and inter-vixel connectors.

For example, Figure 1 shows a visual editor generated by BuildByWire for the visual language of JComposer, a tool for designing and generating CASE tools [7]. This notation includes vixels for tool components and relationships, and connectors between them with associated role and arity information. JComposer components have substructure, including information about the class of the component and its attributes. The end-user clicks on text to edit it or on the arity of a connector Choice to select a different arity. The properties of a selected vixel may be altered through a property sheet; the changeable properties are as defined previously by the notation designer.

A popup menu is associated with each vixel; this includes standard actions, such as deleting the vixel or changing its Z-order. In addition, external code can associate other popup menu items with any vixel to provide for semantic actions managed independently of BuildByWire.

The end-user selects a modal tool from the menu at the bottom of the window; in Figure 1, a Move controller is selected for moving and resizing vixels. There are menu items for the various vixels of the notation, with an associated prototype of that vixel. When a vixel item is selected, the user may click and drag into the work area to create a copy of that prototype and resize it.

The handles of a vixel may be individually or collectively shown or hidden. When a connector tool is selected, the user clicks on the handle of one vixel and drags to another handle in order to place a connector between them. A similar approach is taken to adding constraints over vixels, such as to require two vixels to remain horizontally aligned. As well as outer resize handles, a vixel may have internal handles for manipulating its internal structure or for attaching connectors.

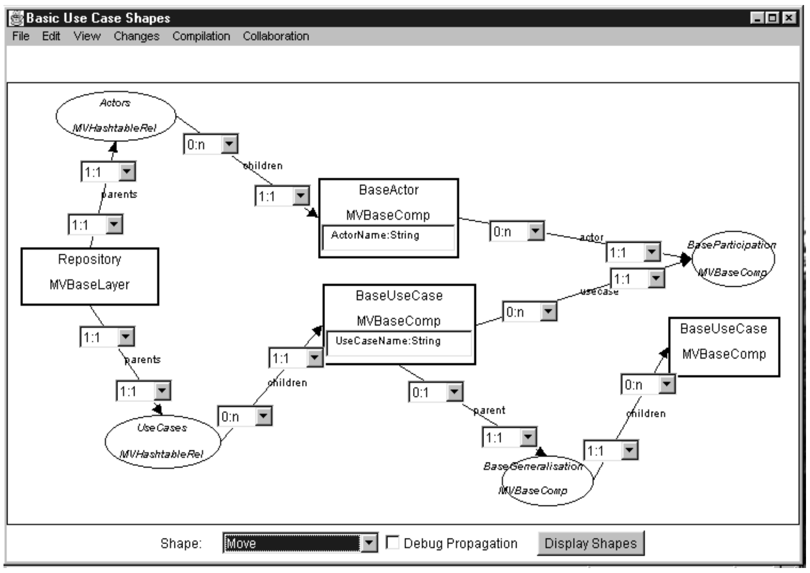


Figure 1. An Editor Generated by BuildByWire: JComposer

BuildByWire was used to develop the user interface for several CASE tools, including a process modelling tool, ER modeller, UML modelling tool and JComposer itself. BuildByWire, as used to provide editors for such tools, proved to have some limitations. A menu was provided to the end-user to select the mode of operation: either a drag tool or a vixel type for click-and-drag creation on the work area. The notation designer had no control over the use of this menu, only over the vixels that were available in it. Designers wanted some control over the generation of suitable tools.

When composing vixels, it was awkward for a designer to add components to a Java container with a layout manager. The layout had no visual form and it was difficult to change layout characteristics such as the number of columns in a GridLayout.

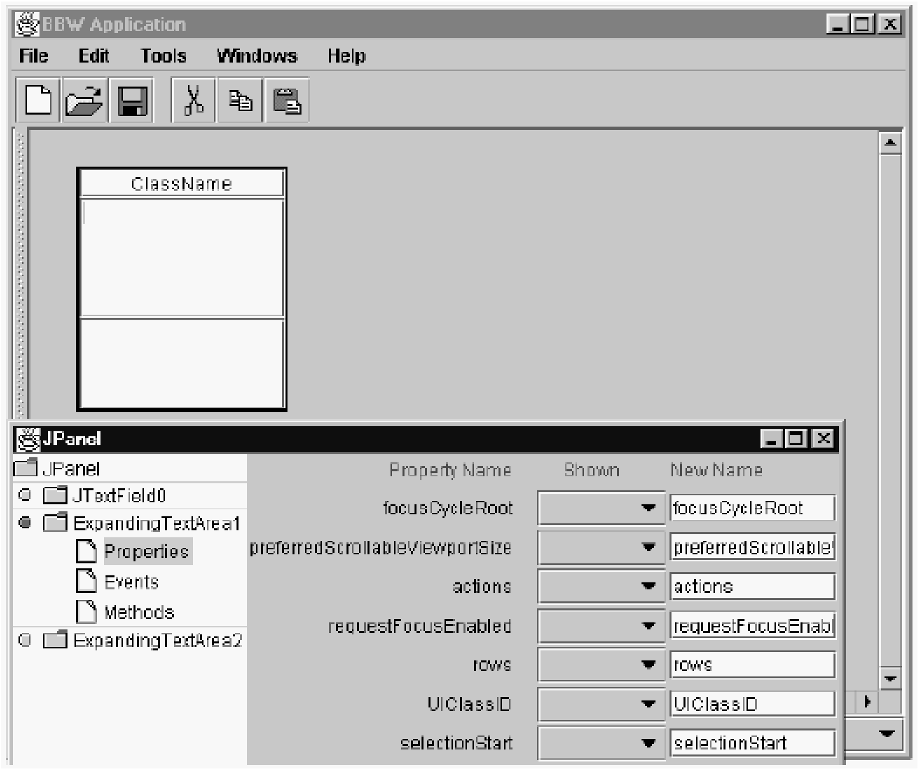
Connectors between vixels could only be attached to the vixel handles. This restricted the attachment points and the resulting diagrams were rather ugly. For example, connectors were inadequate to build UML sequence diagrams.

These limitations necessitated some modifications to BuildByWire, in particular to its facilities for specifying vixel creation, layout and attachment, and its support for reuse of existing JavaBeans. The following section outlines how an editor designer uses BuildByWire to specify editors, and how JavaBeans are both reused and produced by BuildByWire. We then describe some of the usability enhancements we have made to BuildByWire to assist editor designers and users.

3. The BuildByWire Editor Design Process

A designer uses BuildByWire as a meta-editor to define a visual notation and to generate an editor for that notation. Off-the-shelf and customised JavaBean components can be used to compose complex vixels and their interconnections.

A new vixel, such as a UML class icon, is first built as a composite. A JavaBean class for the new vixel is then generated from that composite. A composite is usually defined by taking a standard Java Container, such as a JPanel, specifying a layout and adding lightweight Java components to it, such as a JTextField or JButton. A nested structure may be created by adding containers to the top-level container of the vixel. For example, Figure 2 shows a composite for a UML class that has been constructed from a Container with three elements.



Property Name	Shown	New Name
focusCycleRoot		focusCycleRoot
preferredScrollableViewportSize		preferredScrollable
actions		actions
requestFocusEnabled		requestFocusEnabl
rows		rows
UIClassID		UIClassID
selectionStart		selectionStart

Figure 2. Composition and Creating a Bean

The properties of components added to a composite, such as the font used in a text field or the size and label of a button, are modified using a property sheet. Some of these may need to be fixed at design time, however, so that the end-user cannot change them. In Figure 2, some of the properties of one of the elements of the composite (an ExpandingtextArea object) are shown. The designer selects to *show* those properties to be exported in the generated bean, as well as renaming them where necessary.

Before generating a JavaBean class for the new vixel, it is necessary to define its encapsulation, both through the user interface and through a program (bean) interface visible to external code.

User-interface encapsulation defines how the end-user can change the vixel through the user interface, both by direct manipulation and through the property sheet. Encapsulation of direct manipulation of the elements that make up the vixel is carried out as the composite is constructed. For example, a JTextField may be disabled so that the end-user of the generated editor is unable to alter its contents.

Encapsulation of the properties of the vixel specifies which properties of the originating composite may be changed by the end-user through a property sheet. In addition, some properties of the elements of the composite may also be available for editing. For example, the end-user may be able to change the font of a text field within a vixel. This requires specifying which properties of elements are to be "exported", possibly with renaming to avoid property name clashes (such as between the *text* property of two *JTextFields* within a composite).

Programmatic encapsulation includes those properties of the vixel that may be manipulated from other code. For example, the colour of some component may be altered by external code to signify some semantic state change. The events and methods of the vixel must also be encapsulated. For example, external code may place a listener on a button within a vixel to respond when it is pressed by the end-user, or may add a new component to a container nested within the vixel.

The encapsulation of a composite's properties, events and methods involves generating a new class definition for the composite as a bean, which includes mapping methods to provide access to exported properties (ie getter and setter methods), events (ie addListener and removeListener methods), and methods of composite elements. A related BeanInfo class is also generated to define what properties (and events and methods) of the new bean class are accessible through any bean builder's property sheet.

Once all the vixels of a notation have been composed, the designer requests the generation of the overall editor. BuildByWire generates a Java container bean for the editor, including the list of vixels that may be created, as specified by the designer.

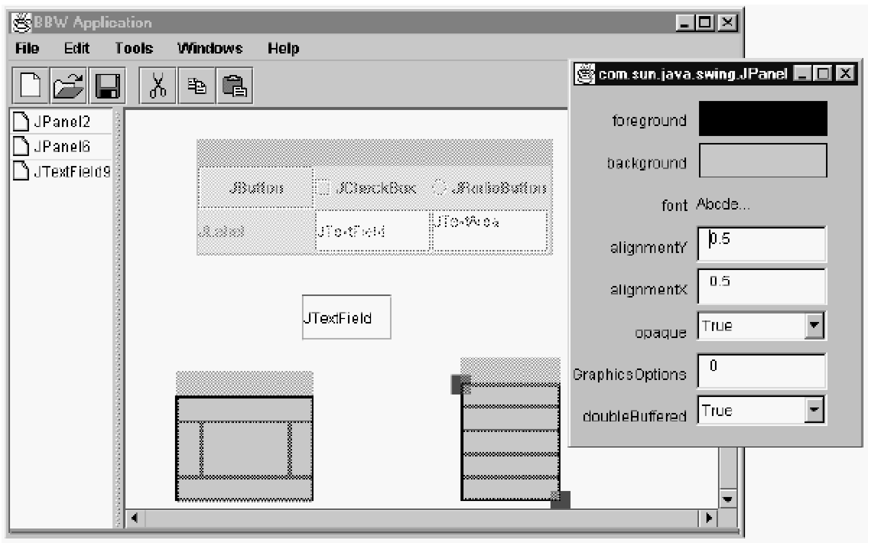
While BuildByWire provides many of the visual drag and drop capabilities of a bean-based development system like JBuilder [17], it is not a programming environment. BuildByWire is not concerned with the semantics of visual notations (such as the consequences of inheritance in UML class diagrams). Semantic processing is provided by other tools. For example, BuildByWire has been used with JComposer to develop a number of multi-view, multi-user visual editors, including an ER modeller, a UML modeller, the visual process modelling language Serendipity-II, and JComposer itself [6, 7, 8, 12].

The beans generated by BuildByWire are deployed by adding them to semantic-processing code, such as that provided by JComposer [6]. A tool designer uses the JComposer visual language to specify repository and view component structures and event-handling behaviour [6, 7]. JComposer uses the JavaBeans introspection mechanism to determine the properties, events and methods of vixel beans. Correspondences are then made between JComposer views and BuildByWire editor beans (panels), and between JComposer view components and BuildByWire vixels. View components may constrain the semantics of BuildByWire editing operations.

JComposer generates code for the JViews API [7] to produce an editor for the visual notation with support for multiple consistent views, multiple users, end-user constraint customisation and task automation facilities. When an end-user edits BuildByWire diagrams, events describing changes to a vixel are captured by its corresponding JViews view component and acted on. When a JViews view component is modified, due to changes elsewhere (such as in another view) it updates the corresponding BuildByWire vixel.

4. CreateThroughs

CreateThroughs provide an extensible lens-based alternative to the BuildByWire modal menu for selecting vixels, as introduced in the previous section.



foreground	■
background	□
font	Abcde...
alignmentY	0.5
alignmentX	0.5
opaque	True
GraphicsOptions	0
doubleBuffered	True

Figure 3. CreateThroughs and DragThroughs

Figure 3 shows BuildByWire in use by a designer. At the top of the work area is a semi-transparent CreateThrough that includes six component types (JButton, JCheckBox, JRadioButton, JLabel, JTextField and JTextArea). By clicking through one of the regions in a CreateThrough, a new component corresponding to that region is created and may then be dragged out to an appropriate size.

Other CreateThroughs are available via the Tools menu to allow other types of component to be created. These may include any lightweight Java Component bean object. CreateThroughs are dragged by a header area at the top.

New CreateThroughs may be created from any container of components. The CreateThrough in Figure 2 was created by adding six components to a JPanel with GridLayout and then selecting a menu option to form the CreateThrough. A semi-transparent image of a copy of the container is made. When the user clicks on the CreateThrough, it clones a copy of the component at that relative position in the copied container.

This makes it trivial for a designer or end-user to tailor-make tools. The components in the Container act as prototypes for new ones and so can have their properties set to a useful state before being added. For example, once all the icons for an editor have been created, they may be added to a container to produce the CreateThrough to be provided to the end-user. We plan to use the same technique to build floating modal tools.

When the designer generates the overall editor, they specify which CreateThroughs are available to the end-user and whether the end-user is able to construct their own (and whether they are able to introduce new beans as new vixels).

5. DragThroughs

A second type of lens, a DragThrough provides a visual form for the layout manager of a container. The bottom two components in Figure 3 are both containers (JPanels); each is overlayed by a DragThrough. The left DragThrough provides a BorderLayout view of the underlying container. Drag regions represent the north, south, etc layout regions of the Container (independent of the elements in it). The right DragThrough provides a GridLayout view, which has been tailored to use one column.

A popup menu selection is made to acquire a DragThrough over a Container. The DragThrough provides options to select the layout manager that's required and to specify layout characteristics, such as the number of rows/columns in GridLayout. Components, such as the JTextField in Figure 3, may be dragged onto one of the DragThrough's drag regions or can be created in place by clicking through a CreateThrough placed over the DragThrough.

Specialised code has to be written for each DragThrough. Several layouts (such as FlowLayout) have a common DragThrough; it simply is needed to specify and alter the sequence of components in the Container. This DragThrough is also used for Swing components that hold a sequence of elements, such as a JComboBox. The most complex layout manager, GridBagLayout, is currently being built.

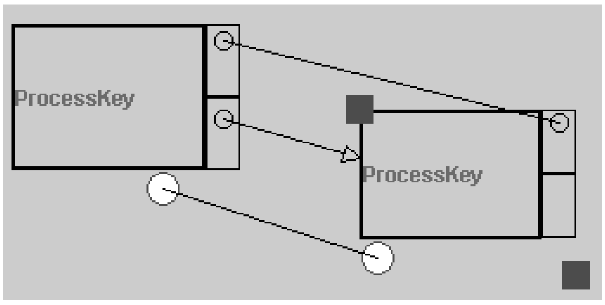


Figure 4. Types of Connections

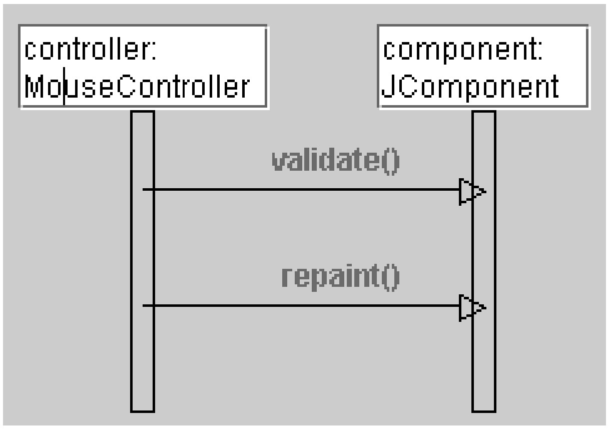


Figure 5. AttachmentRegions for UML Sequence Diagrams

6. Connectors & AttachmentRegions

Many visual notations include connectors between vixels, such as associations between class icons in a UML class diagram. Figure 4 shows several types of BuildByWire connections between two *ProcessKey* vixels. As a *ProcessKey* vixel is moved, the connectors stretch to maintain the connections. There are two means by which the endpoint of a connector may be attached to a vixel. The first type of attachment is to a vixel boundary, as shown by the arrow connector in Figure 4.

The second type of attachment is to a *pin* (visible or invisible). BuildByWire defines an AttachmentRegion container, which may be embedded in a vixel. Associated with an AttachmentRegion is a prototype component, a pin. In Figure 4, for example, a filled circular component acts as the pin in the AttachmentRegion at the bottom of a *ProcessKey* vixel.

When the user clicks on an AttachmentRegion to drag out a connector, the AttachmentRegion creates a pin at the click point by cloning the prototype pin. Similarly, if the other end of the connector is dropped on an AttachmentRegion, a pin is created there. AttachmentRegions don't need to be visible. For example, the *ProcessKey* vixel in Figure 4 has an invisible AttachmentRegion at the bottom. An AttachmentRegion may also have a layout manager associated with it.

Because connectors may be attached to pins, they often cross containment (and hence coordinate space) boundaries. To manage this, they are added to the least upper bound container of the two end-points.

Figure 5 shows how AttachmentRegions can be easily used to build a vixel for a UML sequence diagram, representing method calls between objects. Each object vixel specifies the name of the object and its class in a JTextArea component. This has no border, but contains an AttachmentRegion with an invisible pin below the text. The AttachmentRegion height grows as the vixel is made taller. It has a HorizontallyCentredLayout manager to ensure that each contained pin is horizontally centred in the AttachmentRegion.

Connectors may have attached components, such as text. Such attached components are managed by a LayoutManager, which can ensure that they are positioned relative to either end-point or to the centre of the connector. Alternatively, a multi-directional constraint such as a RelativeConstraint can be used to attach them, as provided in the original Prolog-based BuildByWire [12].

Pins may be moved within the bounds of their AttachmentRegion. In Figure 4, for example, the circular pins can be dragged along the base of the vixels. In Figure 5, the method-call lines may be moved up and down ; horizontal constraints are used to keep an arrow level by keeping its endpoints at the same y-position.

7. Related Work

A variety of approaches have been developed for specifying and implementing visual editors. Common approaches include implementing visual diagramming tools using graphical toolkits, interactively specifying icon appearance and generating icon drawing code, and interactively specifying icon appearance and behaviour, usually interpreting such specifications.

Examples of toolkit-based approaches to visual editor development include Unidraw [17], Garnet [13], Zeus [1], and Amulet [14]. These systems are very flexible and powerful, and allow a great range of visual editors to be developed. However, such approaches decouple specification from final iconic appearance and behavior, and require great time, effort and skill, plus considerable interative refinement, to develop editors. Some reuse of editor components is often possible, but often at considerable effort in managing the complexity of such components. Some limited abstractions similar to BuildByWire attachment regions exist in many of these systems, but no visual DrawThroughs and CreateThroughs. Many such toolkit-based approaches do not support component-based reuse of predefined or standard vixels.

Specialised languages for specifying iconic editor appearance are common, but few have even limited editing and layout behaviour. For example, Chang's two- dimensional iconic parsers [4] allow diagrams which have been free-edited to be parsed and executed. Users typically prefer editors which are "structure-oriented" for most diagramming applications, however, with support from the editor to build iconic forms. Read and Marlin's VSL [15] provides a complex textual language with similar capabilities to BuildByWire for specifying icon appearance, layout constraints and editing behaviour. As it is a textual language, designers can not readily map textual icon and state chart specifications onto their resultant editors.

In earlier work, we developed the Skin icon specification language [18]. This allows complex composition of graphical elements to be specified visually. However, it lacks some forms of layout specification and is rather weak on connector specification. Like BuildByWire, tool semantics are specified separately. Most iconic specification languages don't provide CreateThrough, DragThrough and AttachmentRegion abstractions.

Interactive environments for iconic editor specification allow designers to more easily visualise the resultant iconic forms, and some aspects of editor behaviour. The work of Daberitz [5] generates editors using database schemas and basic icon appearance specifications. Editing functionality and icon layout and connection is limited and automatically generated however, and only very simple composite icons are supported, limiting the flexibility of this approach.

The DV-Centro Framework [2] has some elements in common with BuildByWire. It supports the construction of image elements, consisting of graphic elements and pads. The latter are connection regions used to specify graphical relationships between image elements which are maintained by connecting the image elements using joints. Joints and pads perform a similar role to BuildByWire pins and connectors, specifying constraints, such as alignment, between the associated image elements. However, it lacks elements such as BuildByWire's CreateThroughs and DragThroughs for icon creation and layout.

HotDoc [3] is a tool for constructing compound documents. It has one main element in common with BuildByWire. Like BuildByWire, it provides a visual approach to specifying layout of containers, but does not provide equivalents to CreateThroughs nor flexible AttachmentRegions.

Vampire [10] permits specification of editors (and runtime semantics) for notations such as logic circuits. It attempts to extend the graphical rewrite rule approach to visual notation specification beyond rigid adjacencies to cover connections, geometric constraints and mouse interaction. However, the vixels that can be specified are unsophisticated compared to those of BuildByWire.

Escalante [11] allows visual specification of visual notations via its GrandView visual language. Escalante permits specification of quite complex components, such as pie charts and Kiviat diagrams, using graphical composition techniques. However, it lacks support for reusable components, such as Beans, and lacks BuildByWire's extensibility with CreateThroughs and DragThroughs.

Some VRML world builders such as CosmoWorlds provide sophisticated tools for composing three-dimensional nodes or scenes. However, they lack extensibility and sophisticated direct manipulation of nodes.

8. Conclusions and Future Work

BuildByWire allows a designer to construct by direct manipulation both the notational elements (vixels) of a visual notation and the tools required for an end-user to create, connect, and combine those visual elements. Unlike other similar tools, BuildByWire is designed to operate with standard, reusable JavaBeans parts, so that sophisticated JavaBeans can be incorporated into vixels. BuildByWire generates JavaBeans from composites; these parts may in turn be included in BuildByWire or other beans-based tools.

We have identified several usability limitations in the previous version of BuildByWire. These have been addressed with CreateThroughs, DragThroughs and AttachmentRegions. CreateThroughs provide a way of adding tailor-made tools; new CreateThroughs can be created from any container of components. CreateThroughs support more flexible, extensible and easier to use creation of vixels by editor designers and users. DragThroughs make the layout of a container visible so that components may be more easily added to it. Editor designers use DragThroughs to place vixels within containers in appropriate ways. AttachmentRegions may be included in vixels, along with prototype pins, to allow for flexible connections between vixels. This allows editor users to connect their vixels in many different ways, as appropriate to their visual notation's requirements.

Further work is underway to make BuildByWire more generally applicable. We plan to generate modal tools (floating and docked) in a similar way to CreateThroughs. Gesture-based creation techniques may well be incorporated in a similar manner. Specialised components for managing poly-line connectors and rotated text on connections are needed for some notations. DragThroughs are currently being built for GridBagLayouts, menus and hierarchical viewers.

References

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

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

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

[4] Chang, S.K., Costagliola, G., Orefice, S., Tucci, M., Tortora, G., and Polese, G., “A 2D Interactive Parser for Iconic Languages,” in *Procs of the 1992 IEEE Workshop on Visual Languages,* IEEE CS Press, Seattle, WA, 1992, pp. 207-213.

[5] Daberitz, D. and Kelter, U., “Rapid Prototyping of Graphical Editors in an Open SDE,” in *Procs of 7th Conference on Software Engineering Environments,* IEEE CS Press, Netherlands, 1995, pp. 61-73.

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

[7] Grundy, J.C., Hosking, J.G., and Mugridge, W.B., “Visual specification of multi-view visual environments,” in *Procs of 1998 IEEE Symposium on Visual Languages,* IEEE CS Press, Halifax, Nova Scotia, Canada, 1998.

[8] 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, 1998.

[9] Sun Microsystems, “Java Beans 1.0 Specification”, 1996.

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

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

[12] Mugridge, W.B., Hosking, J.G., and Grundy, J.C., “Towards a constructor kit for visual notations,” in *Procs of OZCHI'96,* IEEE CS Press, Hamilton, New Zealand, Nov 25-28 1996, pp. 169-176.

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

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

[15] Read, M. and Marlin, C., “Specifying and generating program editors with novel visual editing mechanisms,” in *Procs of the 10th Conference on Software Engineering and Knowledge Engineering,* KSI Press, San Francisco, USA, 1998, pp. 418-425.

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

[17] Vlissides, J.M., “Unidraw - a framework for building domain-specific graphical editors”, *ACM Transactions on Information Systems*, 8, 3, pp 237-268, July 1990.

[18] Hosking, J.G., Mugridge, W.B., Fenwick, S, and Grundy, J.C., 1995: “Cover yourself with skin”, *Procs of OZCHI'95*, University of Wollongong Nov 1995, pp 101-106.

[19] http://www.cosmosoftware.com