Title: Towards a Constructor Kit for Visual Notations

Warwick B. Mugridge, John G. Hosking, and John Grundy§
Department of Computer Science, University of Auckland
Private Bag 92019, Auckland, New Zealand
rick@cs.auckland.ac.nz

§Department of Computer Science, University of Waikato,
Hamilton, New Zealand

Contact Person: Dr Rick Mugridge
Department of Computer Science, University of Auckland
Private Bag 92019, Auckland, New Zealand
rick@cs.auckland.ac.nz

In Proceedings of OZCHI'96, Nov 24-27 Hamilton, New Zealand, IEEE CS Press.

Towards a Constructor Kit for Visual Notations

Abstract

BuildByWire is a constraint-based environment for specifying and implementing visual notations for environments developed using the MViews framework. Previous approaches to user-interface construction based on constraint propagation approaches have been limited We describe a new approach that avoids some of these difficulties through the use of the multi-directional constraints of Snart, an object-oriented extension of Prolog.

1. Introduction

The aim of the BuildByWire project is to design, build, evaluate and evolve a construction kit for visual notations. This work is motivated by our development and use of MViews, a general-purpose framework for implementing software development environments that maintain consistency between multiple graphical and textual views [1]. For example, MViews has been used to construct: SPE, the Snart Programming Environment [2]; Cerno, a program visualisation system [3]; ViTABaL, a tool-based system [4]; and Serendipity, a work-flow modelling system [5]. In such development environments the user is able to manipulate specialised visual notations in graphical views. For example, in SPE, the user may add new classes, change superclass relationships, rename class attributes etc; these changes are reflected in other views, including program text views.

The MViews framework has strong support for consistency management (including for multi-user collaboration) but is somewhat lacking in high-level support for user interface construction. In the MViews-based environments that have been constructed so far, the program code that allows the end user to manipulate notations in graphical views is constructed manually. BuildByWire aims to eliminate much of this effort by providing a high-level tool for creating notations and defining how they may be manipulated by the end user. The prototype of BuildByWire presented here is itself based on the same metaphor: that of creating and manipulating visual notations; a longer-term aim of our work is to use BuildByWire to implement its own user interface.

BuildByWire allows a visual notation designer to compose the elementary graphical elements that make up a visual notation and to specify: how those elements may be related, how they change under transformations (such as resizing) and how changes are to be mapped to and from an underlying representational layer. The prototype is implemented in Snart, an object-oriented extension to Prolog that incorporates high-level, multi-directional constraint expressions [6,7,8].

The remainder of the paper is organised as follows. In the next section, we introduce requirements for a visual notation constructor kit. This is followed by the design of BuildByWire and a description of the first prototype. The next section describes how Snart constraints are used to implement some of the features of BuildByWire. This is followed by a description of related work, discussion, and conclusions.

2. Requirements of a Visual Notation Constructor for MViews

In order to determine the requirements for a visual notation constructor kit, it is instructive to examine the visual notations and user interfaces of existing tools built using MViews. Fig. 1 shows SPE in action, with three graphical views representing aspects of the design of an object-oriented program. The annotated boxes represent classes in the design while the arcs represent relationships, such as inheritance or client-supplier, between the classes. A tool palette at the left of each window allows users to select tools to manipulate the design.

A visual notation has two aspects: the visual form, ie the rendering of components of the notation, and the manipulation semantics, or the way in which the user can interact with the components to manipulate a diagram. A constructor kit needs to be able to specify both of these aspects, together with the way in which the visual components interact with the underlying application data structures. Following are examples of each of these aspects that can be seen in SPE.

The tool palette consists of a collection of iconic renderings, laid out regularly, but fixed in position. The icons are sensitive to mouse clicks, inverting their rendering to indicate selection. Selection indicates a change to the editing "mode" of the window, ie the manipulation semantics of the window changes according to the selection.

Class icons are composite graphical objects, displaying the class name and class features as text, bordered by a round box and a line separator. Their manipulation semantics differs with the tool mode. With the select tool, for example, the class icon may be dragged around the window to change its location. Dragging the class icon causes all of its constituent components to move in unison. Selection of a class icon is indicated by addition of four handles to the icon. The handles, however, cannot be used to resize the icon.

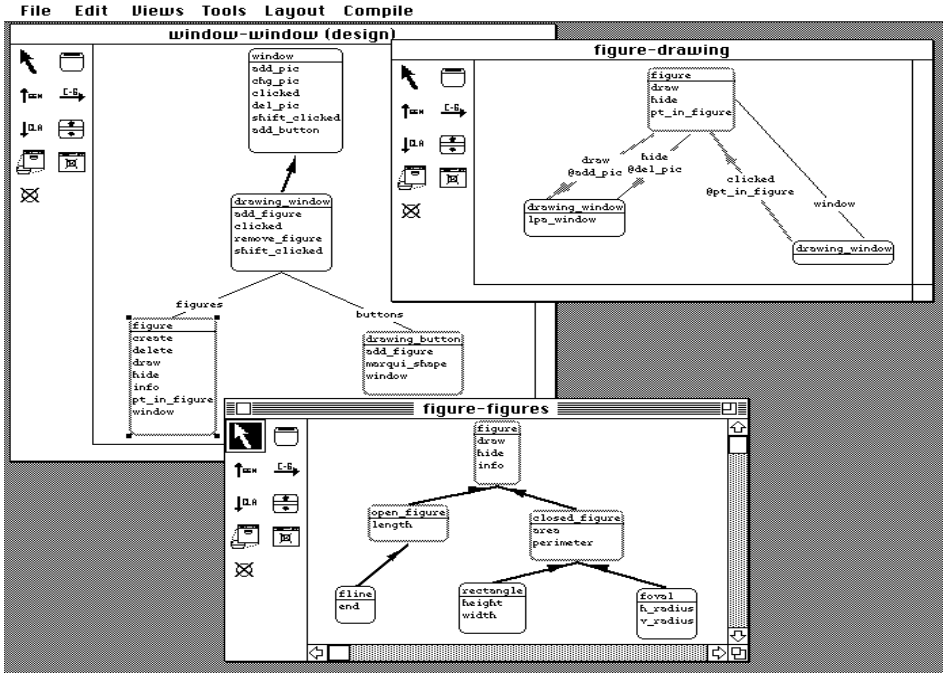

Figure 1. SPE in action

Internally, a class icon has a number of "click points", as shown in Fig. 2. Clicking in these with the select tool permits navigation to other relevant views, such as other views containing the class or feature or the textual implementation view of the class or feature. The visual manipulation must thus be translated into a service request in the underlying application. Using other tools, different semantics result. For example clicking on a feature text using the feature tool permits attributes of the feature, such as its name and type, to be edited. Clicking at the bottom permits a new feature to be added to the class. Both of these types of change result in a change in the icon rendering to reflect the modifications made. Dragging from a class icon with the inheritance tool causes an inheritance arc to be added. A new class icon may also be added.

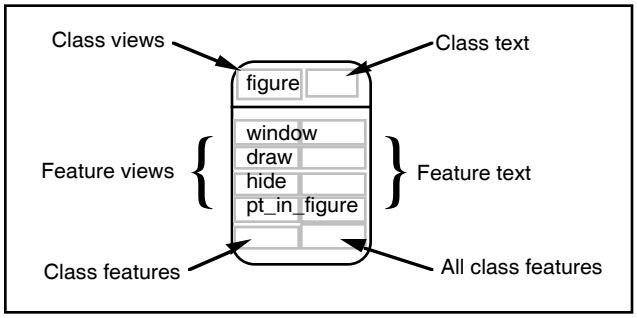

Figure 2. Structure of a class icon in SPE

Relationship arcs are also composite graphical objects, consisting of a styled line and optionally a triangular arrowhead, and/or a textual annotation. Relationship arcs move with the class icons they are attached to. As an arc moves, the arrow head and textual annotation maintain their original *proportional* distances along the arc. Arcs may also be manipulated by the select tool using drag handles at each end. These permit the ends of an arc to be moved, but the movement is constrained to be around the perimeter of the attached class icon. Some of the other tools permit interaction with arcs; for example clicking on an arc with the hide tool causes the arc to be removed from the view, but not the underlying application database.

MViewsDP, a visual dialog box editor, introduces some additional manipulation semantics. Forms are constructed as rectangular boxes, with fields (annotated rectangular boxes) inside them. Fields and dialog boxes may be resized, using handles. Fields may be moved around inside the form by dragging. However field resizes and moves are constrained to keep all of the field inside the dialog box. It is also desirable to avoid overlap of fields within the dialog box.

The structure of all the visual notations used in tools built with MViews have thus been based on a mixture of graph and simple containment notations. In the existing framework, both the visual form and the manipulation semantics must be defined procedurally, using textual code. Some elements are reusable. For example, the connector arcs have a fairly common form in all applications, and simple parameterisation based on line style, and presence or absence of annotations and arrow heads is sufficient. However more complex, and hence more interesting, components require substantial input from the framework user. For example, the SPE class icon requires several pages of code to define. In addition, the visual form and manipulation semantics are defined separately, reducing the understandability.

Our aim with BuildByWire is to eliminate as much as possible the writing of procedural code to define such graphical components, by permitting a declarative graphical

specification. This approach emphasises composition of basic components, including appropriate composition semantics such as composed objects moving together, and visual specification of manipulation semantics using constraint tools.

3. Design of BuildByWire

In this section we describe a design for BuildByWire, focussing on the way in which the visual form and manipulation semantics are specified. The mechanism for mapping changes at the user-interface level of the tool into changes at the underlying representation layer, and vice versa is reasonably straightforward and has been addressed in other work [9], so will not be discussed further here. We initially focus on the specification of composite figures, considering more complex aspects of manipulation semantics later.

Composite figures are constructed from component figures (both built-in atomic figures and other composites) visually by direct manipulation. Three main categories of atomic elements are provided:

- *figures*, such as boxes, ovals, lines, text, images, pop-up menus and buttons. These have associated properties (size, position, filling, line width, font size, etc) and "reshaping" handles (for resizing, rotating, shearing, etc).
- *wires*, which impose constraints on the properties of figures. Some wires, such as relative and proportional wires, also introduce *pins*, which may be used to attach further wires (discussed below).
- *containers*, which organise collections of figures, such as vertical lists, organisational chart structures, card stacks (which display one of several sub-figures, in order to show state changes) and other layout managers. A variant is also used to transport sets of figures around.

Using wires, the relative (cartesian coordinate) position of component figures can be specified, as well as the manner in which repositioning and reshaping (resizing, rotating, swivelling, shearing) of one component affects another.

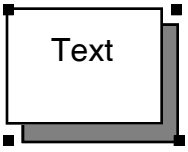

Figure 3. A composite graphical element

For example, Fig. 3 consists of a text field contained within a box with a drop shadow and resize handles positioned at the boundary of the whole figure. This figure is constructed from several atomic figures, including a text field, two boxes, two *relative* wires, four *equality* wires, and four *proportional* wires. The equality and relative wires are used to position the shadow relative to the main box, while the proportional wires position new handles relative to the composed figure.

The pin of a relative wire is offset from a "home " pin (a handle or another pin). It may be moved relative to the home pin, changing its offset, but when the home pin is moved, the offset is maintained. This allows for offsets to be defined graphically, such as the offset of the shadow in Fig. 3. An equality wire, dragged between two pins, imposes a constraint so that the two pins remain superimposed, regardless of which is moved. The pin of a proportional wire is positioned relative to *two* home pins. It may be dragged around to reposition it relative to the home pins. If one of the home pins is moved, the proportional pin moves to retain its relative proportional position between the two home pins. For example, if the new pin is positioned half-way between the home pins, it will remain half-way between, regardless of the movements of either of the home pins. This may be used, for example, to position a text label two-thirds of the way along an arc. In Fig. 3 proportional pins maintain the extra handles proportionally to the top left and bottom right corners.

Fig. 4 shows how Fig. 3 is constructed. In Step 1, the box tool is selected and a box is interactively dragged out; this will eventually be the shadow box. In Step 2, the relative pin tool is selected and two relative pins, offset from the corners of the box by the same amount, are added (by dragging from the box handles). These pins are used to define the offset to the text box that is added in Step 3. In Step 4, the equality wire tool is selected and equality wires are connected by dragging between the corners of the text box and the relative pins. The combined effect of these is to constrain the text box to be at a defined offset from the shadow box and the size of the shadow box and text box to remain equal. Step 5 adds proportional pins on each side of the main box, between the top and the bottom, together with the text field. In Step 6, equality wires between the bottom text handles and the proportional pins position the text in the top half of the main box. The two extra handles at the top-right and bottom-left are created in step 7, using proportional pins, with the top left handle of the text box and the bottom right handle of the shadow box used as home pins (note that relative pins could have been used instead). The text box is coloured white and the shadow grey in step 8, by manipulating the properties of those figures.

Finally, in step 9, the composite figure definition is completed by hiding pins and handles used purely for internal construction. All the handles of the original text area are invisible in the final figure, as are three of the handles of each of the two original boxes. Also hidden are the constraints that ensure that the two boxes remain the same size; hence the end user is unable to alter the relative distance between the main box and its shadow. However, the end user is able to alter the text and to resize the figure; the boxes grow larger if necessary, to accomodate any text changes, the text is recentred in the space available if the figure is resized, while the text's font size will be reduced if the user makes the main box size too small.

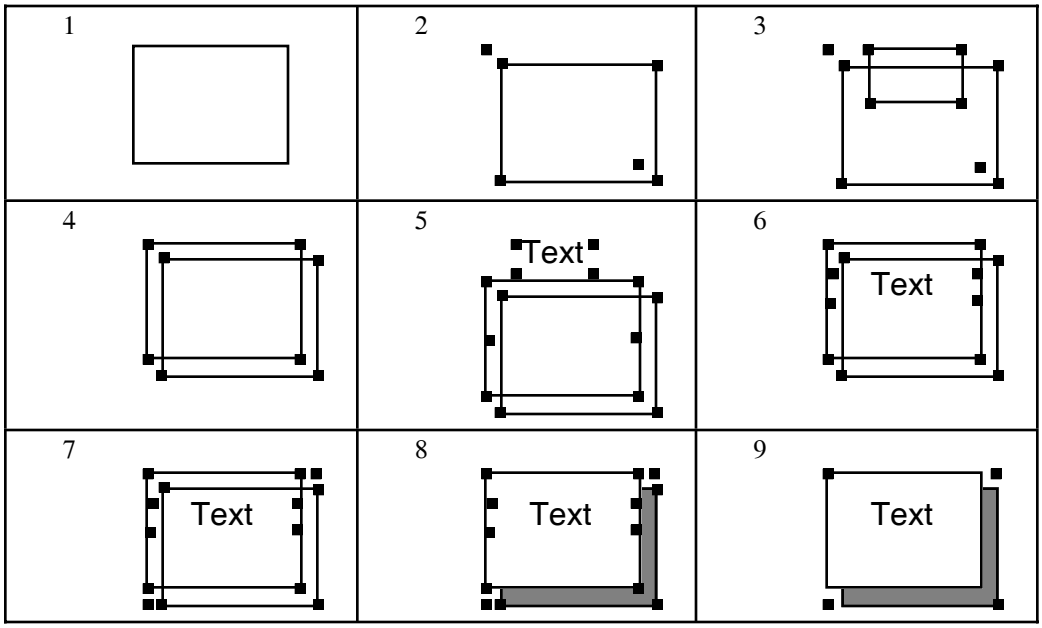

Figure 4. Steps in composing a shadow box with text

The resulting new figure may be added to the set of atomic figures provided by BuildByWire and used in the construction of more complex figures. As the standard transformations may be applied to the new figure, the way in which the figure was composed will determine how any transformation will be applied to the components of the new figure. For example, a translation of the the shadow-box figure of Fig. 4 will translate all of its components by the same amount. However, sometimes such standard transformations are not what is required.

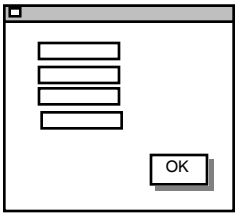

Figure 5. Limiting the effect of reshaping

Consider the development of the dialog box shown in Fig. 5, which has an OK button constrained by a proportional pin to be proportionally offset from the bottom right hand corner. The user interface this component is being developed for is to provide two different ways of resizing the dialog box. The first, or *shrink*, operation, uses the standard approach: resizing the dialog box as a whole also resizes its internal components, ie dragging on the bottom right hand corner causes a proportional resizing of the OK box. In the second, or *resize*, operation, the dialog resize results simply in a change of position of the OK button rather than a proportional resizing, ie the components of the dialog are squashed together (or pulled apart) but retain their original size. There is thus a need to have different manipulation semantics depending on the *reason* for the manipulation (resize or shrink).

This is achieved by propagating not only the changes to values along wires, but also the reason for the change. At various points along the connections, it may then be necessary to map from one reason to another to achieve the appropriate semantics. For example, Fig. 6 shows the reason mapping that occurs along the equality wire joining the proportional pin to the OK box corner. If the proportional pin location changes because of a shrink, this is converted to a modification of the corner of the OK box with a resize reason, resulting in a resizing of the box. However, if the proportional pin location changes due to a move or resize reason, the box corner is changed with a move reason. This causes the whole OK box to shift location, rather than resize. The ability to manage this degree of flexibility in the specification of manipulation semantics is due to the unique nature of the Snart dual constraint propagation mechanism used to construct BuildBy Wire. This is discussed further in the next section. Similar reason mapping tables are required for specifying the drag semantics of composite objects (eg does dragging a component shift it, or attach a new arc to it).

Reason in	Reason out
shrink	resize
resize	move
move	move

Figure 6. Reason mapping

Rotational constraints and relative polar coordinates can also be specified with wires. Fig. 7a shows a rotated box which contains text. The rotation is fixed here and cannot be altered by the end user. On the other hand, Fig.7b shows a directed arc made up of three lines and some text.

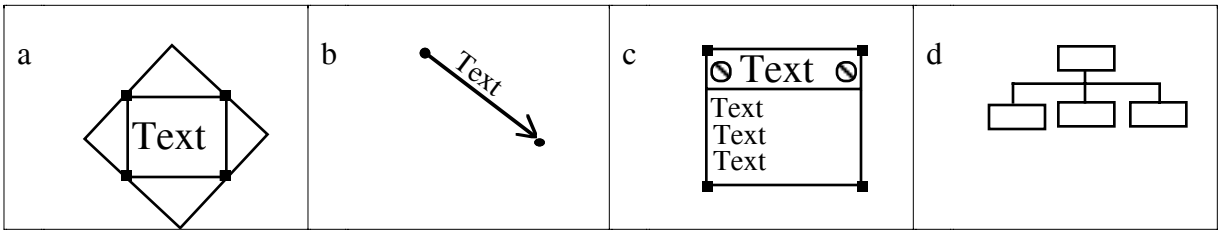

Figure 7. Some composite graphical elements

This arc is to be connected between two figures (nodes) in the final application, so the orientation of the arc needs to change as those nodes are moved. Hence the orientation of both the text and the lines making up the arrow head are based on the orientation of the main line. When either of the connected nodes are moved by the end user, the arc is to resize; however, if the arc is moved, the connected nodes are to move. Hence the connections that are made between them have to include reason mappings, as with the previous example with the OK button. In addition, there are likely to be restrictions on what connections can be made with such arcs. Our current approach is to assume that the underlying implementation of the notation handles the rejection of incorrect connections; another approach would be to define restrictions on the connection points themselves.

Containers organise sets or sequences of elements. For example, Fig. 7c includes a vertical list of text that can be altered by the end user (or by the underlying tool), as well as two areas at the top which act as buttons (these need not be visible). Fig. 7d shows a container that organises a collection of elements into a hierarchy.

4. An Initial Prototype

An initial prototype of BuildByWire has been developed in Snart. A screen dump is shown in Fig. 8, in which the user is in the process of building a shadow box with text. The prototype permits the composition of elementary graphical components for use by the end-user of the visual notation. A subset of the wires, pins, containers and atomic figures have been implemented using Snart's multi-directional constraints and *demons*. These are shown in the tools on the left hand side of the window.

Implemented figures include generic boxes (which can be "morphed" between three basic types: box, oval and text), lines (which can be "morphed" between line, arrow and double-arrow). Figures have handles, which are used for resizing and which can be used to constrain the figure; other forms of reshaping have not yet been implemented.

The following types of wire have been implemented: relative (with associated pin); equality; horizontal alignment (dragged between two pins, they impose a constraint ensuring that the y-dimension of the two pins remains the same, regardless of which is moved); vertical alignment; arc (to connect a line between two pins, to be maintained when either or both pins are moved); and owner (which connects two pins to combine the effect of a relative wire and an equality wire: when the first pin moves, the second pin moves with it; however, the second pin moves independently of the first).

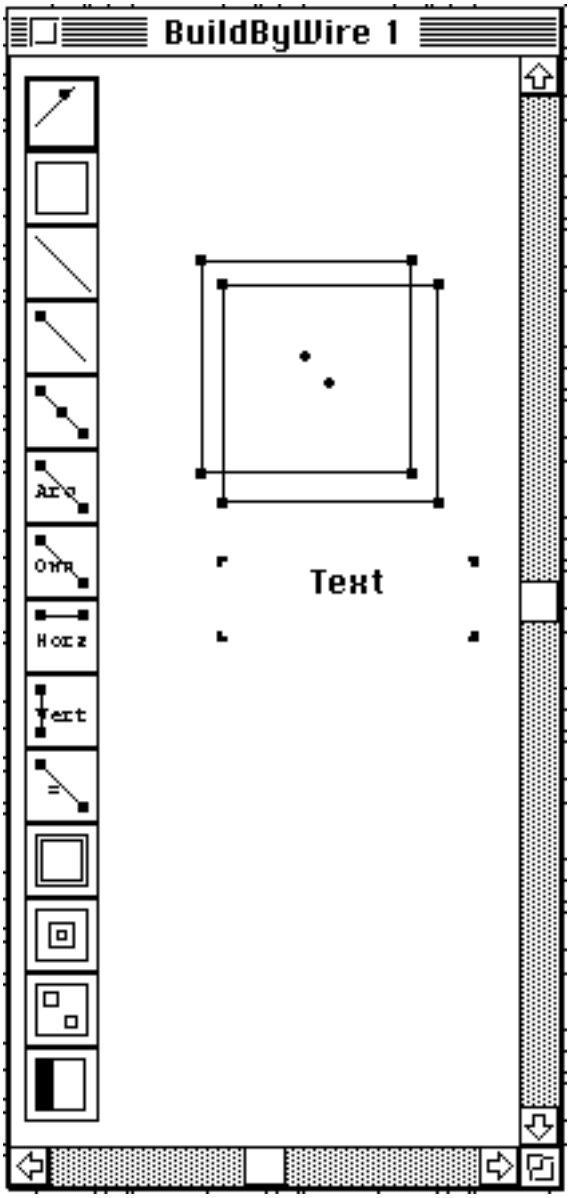

Figure 8. Initial prototype of BuildByWire

BuildByWire containers include "open" containers, teleporters and tables. Once a figure is placed inside an "open" container, the figure moves along with the container. A teleporter has a window associated with it. Any figure dragged into the teleporter appears in that

window[1]. A table holds a horizontal or vertical list of elements. New elements can be added to a table by dropping a component onto the head of the table. Components can also be added to or removed from existing elements of the table.

Constraints are thus used to compose figures and to specify how such compositions change when they are resized and when they are connected to other components. Constraints will also be used to maintain consistency between visual objects and their underlying representational objects. The reflexive nature of Snart means that new constraints (types of wire) can be defined on-the-fly, allowing BuildByWire to be easily extended. A demon is a piece of procedural code that is executed when there are changes to the variables that is references. Demons are used for redrawing figures that have changed and to handle processing that is beyond the constraints (such as handling containers).

This initial prototype has already shown some of the benefits and limitations of our approach. The basic elements are straightforward to use, although they are rather limited in their scope. There is a clear need for a wider range of building blocks. The architecture of the prototype enables additional building blocks to be added dynamically, as it is likely that notation designers will want to add their own figures and constraints. Once a wire has been added, it's currently not possible to remove it. Wires therefore need a visual form so that they can also be manipulated. Such a visual form for constraints is provided in the PlanEntry system [9]. Another approach is to have a "Show wires list" dialog which lists the wires attached to a figure and allows deletion of any of them.

Reason mappings are hard-coded in the wires at present; due to the large number of mappings possible, we plan to make the selection of reason mappings more dynamic, leading to changes in the implementation of constraints in Snart. Lack of speed is the major problem with this prototype; the slow response makes it difficult to compose and use complex figures. This is due in large part to the implementation of constraint propagation in Prolog, which is not well suited to the task. Hence we are extending Java with the constraint mechanism of Snart. BuildByWire will then be implemented and extended in this new language; this will markedly improve both the speed and accessibility of this work to others.

5. Snart Constraints

The initial prototype was developed in Snart, an object-oriented extension of Prolog [2] that has been extended further with multi-directional constraints [6,7]. Constraint expressions are expressed directly in the language. Snart constraints have been used to build two other applications: PlanEntry, a multi-view building plan drawing package [9], and DrawByWire, a drawing package [8].

Snart introduces a new approach to resolving the ambiguity of a multi-directional constraint involving more than two variables. *Directions* specify locally how constraints are to be interpreted, independent of other constraints. The directions need not be given explicitly; default directions are provided for constraints without them. For example, consider the following constraint.

a + b = c

The default directions for this constraint are given explicitly as follow:

a + b = c with (a =>b, b => a, c => a)

The directions specify that when the value of a changes, the constraint is resatisfied by reevaluating the value of b (ie b := c - a), while if the value of b or c change, a is reevaluated (ie a := c - b).

Directions can also specify several interpretations of a constraint, depending on the *reason* for a change. *Dual-propagation* is used to propagate both values and reasons through a constraint network. Reasons in Snart constraints have strongly influenced the design and initial implementation of BuildByWire, providing for interaction between visual elements of a notation which cannot be handled by any other constraint system.

For example, the class arc defines a line that remains connected between two pins (start and end), regardless of which is moved. The constraints of this class are defined in Fig. 8. The first constraint defines a simple equality between the x coordinates of the orgin (top-left corner of the arc itself) and start, along with a direction that defines a mapping. If the origin of the arc is moved, this will move the start pin. If the position of the start pin is changed, this leads to a resize change of the origin of the line. This is due to the mapping in the direction "start@x => resize(origin@x)", which means that a change to the value of start@x, regardless of the current reason, will be mapped to a change to the value of origin@x, with reason resize. So the line is resized, regardless of the reason for the shift in position of the start.

The class *proportional_pin*, as shown in Fig. 9, makes use of constraints to maintain a pin in relative position between two other pins (otherPin and owner). The pin is at position (x,y), while the current offset ratio of the pin is kept in the vector offsetRatio. If the pin itself is directly moved (with reason *move_pin*), the offset ratio is altered; otherwise a change to the position of either otherPin or owner leads to a change in position of the proportional pin.

[1] Constraints are used in the underlying implementation to connect handles to boxes, lines, and containers. A bug in the early implementation of teleporters meant that when one of its handles was moved to reduce its size, the handle itself was teleported to the associated window!

```
class(arc, inherits(morphic_line,wire),
     % inherits start, end, origin, corner & line drawing
 ...
 constraints(
   origin@x = start@x and_with (start@x=>resize(origin@x)),
   origin@y = start@y and_with (start@y=>resize(origin@y)),
   corner@x = end@x and_with (end@x=>resize(corner@x)),
   corner@y = end@y and_with (end@y=>resize(corner@y))
 )).
```

Figure 8. Constraints in class arc

```
class(proportional_pin(otherPin:pin),
 inherits(pin), % inherits owner (a pin) from pin
 features(offsetRatio:vector, ...),
 constraints(
   x = offsetRatio@x * (otherPin@x - owner@x) + owner@x and_with
          move_pin(x =>offsetRatio@x),
   y = offsetRatio@y * (otherPin@y - owner@y) + owner@y and_with
          move_pin(y =>offsetRatio@y)),
 ...).
```

Figure 9. Constraints in class proportional_pin

6. Other Work

Several systems for building user interfaces have been based on constraint processing, most using constraint propagation. An early example is ThingLab, which was developed in Smalltalk, with methods used to define how constraints are to be satisfied [10]. Garnet [11], Rendezvous [12] and Escalante [13] are user-interface building systems that are based on uni-directional constraints. The ideas of Garnet have been ported to Amulet, which extends C++ with constraints [14].

These systems are somewhat limited in expression by uni-directional constraints, and are unable to handle different change semantics, as provided by Snart's reasons. In addition, they do not offer the same level of interactive specification as that provided by BuildByWire.

Constraint hierarchies are used in the programming language Kaleidoscope [15] and have been incorporated into a version of Garnet. The relative strengths of constraints are used to remove ambiguity and to allow for over-constraint. Some user-interface development has been carried out using Kaleidoscope. However, due to the non-local nature of hierarchical constraints, "... large constraint networks can be difficult to construct and understand" (Sanella, 1994). In contrast, Snart constraints can be understood locally and Kaleidoscope could only handle reasons in an indirect and clumsy fashion. The main advantage of constraint hierarchies over the Snart approach is that they handle over-constrained networks better.

Skin [17] takes a visual language approach to developing visual components in a system for debugging object-oriented programs. In Skin, the composition of composite figures is defined in a functional language style, rather than by direct manipulation.

QOCA is a constraint solving toolkit that has been used to develop a graphical editing framework [16]. While in many ways the QOCA approach to constraints is more general than our own, it is unclear whether it handles some of the problems that we have addressed. It is also unclear whether it suffers from the fundamental problems of general constraint-solving approaches: efficiency, knowing when constraints are beyond the capabilities of the solver, and in understanding and debugging the constraints when they go wrong [18].

7. Conclusions

BuildByWire has been designed as a constructor kit for visual notations in MViews-based systems. It offers a new direct-manipulation approach to composing graphical figures, defining how they may be reshaped and connected, and defining how changes are to be propagated through those connections. It offers several advantages over existing approaches. These include the use of direct manipulation to define the appearance and change semantics of composite figures. In addition, it is based on a very flexible and "extensible" constraint-based implementation language that avoids limiting the notation designer to a fixed set of building blocks.

While an initial prototype has shown the benefits of our approach, performance is poor. This has been due to using Prolog as our means of implementation. Hence the next prototype of BuildByWire will be built in Lava, a constraint-propagation extension of Java whose implementation is almost completed. Initial timings show a performance improvement approaching 1000 times over the Prolog implementation, making it practical to produce a responsive BuildByWire system.

In addition, further work is required to extend the basic building blocks; we are currently redesigning the basic elements to include a smaller set which can be combined in more powerful ways. Visual forms for the constraints are also needed. For the latter, we plan to explore the use of

movable filters [19] to avoid clutter. Using this approach, a lens may be moved over areas of a composite in order to show the detail of the constraints that have been imposed. Having a visual form of constraints will also mean that constraints may be imposed on constraints; for example, the offset of two relative pins could be constrained to be equal.

Any constraint approach is limited by the constraints that can be realistically handled. Even with dual propagation, our approach is limited by the expressiveness of propagation mechanism. More general constraint solvers are also limited in the range of constraints they can realistically solve. Hence we plan to explore further the use of demons, with associated procedural code, to avoid the limitations of a purely constraint based approach.

Acknowledgments

The first two authors acknowledge the financial assistance of the Auckland University Research Committee.

References

[1] Grundy, JC, Hosking, JG, and Mugridge, WB, 1996, "Supporting flexible consistency management via discrete change description propagation", to appear in Software - Practice & Experience.

[2] Grundy JC, 1993. "Multiple textual and graphical views for interactive software development environments", PhD thesis, Department of Computer Science, University of Auckland, New Zealand, 1993.

[3] Grundy JC, Hosking JG, Fenwick S, and Mugridge WB, Connecting the pieces. Chapter 11 in Visual Object-Oriented Programming, Burnett M, Goldberg A, Lewis T, Eds, Manning/Prentice-Hall, 1995.

[4] Grundy JC and Hosking JG ViTABaL: A Visual Language Supporting Design By Tool Abstraction. In Proceedings of the 1995 IEEE Symposium on Visual Languages, IEEE CS Press, Darmsdart, Germany, September 1995, pp. 53-60.

[5] Grundy JC, Hosking JG "Serendipity: integrated environment support for process modelling, enactment and improvement," Working Paper, Department of Computer Science, University of Waikato, 1996.

[6] Blackmore S, Hosking JG, Mugridge WB, 1994. "Dual propagation in a multi-paradigm programming language", Report No. 86, Department of Computer Science, University of Auckland, New Zealand, 1994, 8pp.

[7] Mugridge WB, 1995. "Snart95 Reference Manual", Department of Computer Science Report, University of Auckland.

[8] Mugridge WB, Hosking JG and Grundy JC, 1995. "An object-oriented programming language augmented with mult-directional constraints", Department of Computer Science Report, University of Auckland .

[9] Hosking JG, Blackmore S, Mugridge WB, 1994. "Objects and constraints: a constraint based approach to plan drawing", in Mingins, C. and Meyer, B. Technology of object-oriented languages and systems TOOLS 15, Prentice Hall, Sydney, pp 9-19, 1994.

[10] Borning A, 1981. "The programming language aspects of ThingLab, a constraint-oriented simulation laboratory", ACM Trans. Programming Languages and Systems, **3**(4), pp353-387.

[11] Myers B, Giuse D, Vander Zanden B, 1992. "Declarative programming in a prototype-instance system: object-oriented programming without writing methods", OOPSLA'92, pp184-200.

[12] Hill RD, 1993. "The Rendezvous constraint maintenance system", UIST'93.

[13] McWhirter JD and Nutt GJ, "An environment for the rapid construction of visual language applications", Procs. IEEE 1994 Workshop on Visual Languages, VL'94.

[14] McDaniel R and Myers BA, "Amulet's dynamic and flexible prototype-instance object and constraint system in C++", Report CMU-CS-95-176, School of Computer Science, Carnegie Mellon University.

[15] Freeman-Benson B, 1990. "Kaleidoscope: mixing objects, constraints and imperative programming", ECOOP/OOPSLA'90, pp77-88.

[16] Helm R, Huynh T, Marriot K, and Vlissides J., "An object-oriented architecture for constraint-based graphical editing", *Advances in Object-Oriented Graphics II*, Springer Verlag, 1993.

[17] Hosking, J.G., Mugridge, W.B., Fenwick, S, and Grundy, J.C., 1995: "Cover yourself with skin", OZCHI'95, Wollongong Nov 1995, pp101-106.

[18] Meier M (1995) Debugging constraint programs. In Proc. Principles and Practice of Constraint Programming, CP'95, LNCS 976, pp 204-221.

[19] Stone MC, Fishkin K and Bier EA, "The moveable filter as a user interface tool", *CHI94*, pp306-312.