GRAPHICAL TECHNIQUES IN A SPREADSHEET FOR SPECIFYING USER INTERFACES

Brad A. Myers

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

ABSTRACT

Many modern user interface development environments use *constraints* to connect graphical objects. Constraints are relationships that are declared once and then maintained by the system. Often, systems provide graphical, iconic, or demonstrational techniques for specifying some constraints, but these are incapable of expressing all desired relationships, and it is always necessary to allow the user interface designer to write code to specify complex constraints. The spreadsheet interface described here, called C32, provides the programmer with the full power of writing constraint code in the underlying programming language, but it is significantly easier to use. Unlike other spreadsheets tools for graphics, C32 automatically generates appropriate object references from mouse clicks in graphics windows and uses inferencing and demonstrational techniques to make constructing and copying constraints easier. In addition, C32 also supports monitoring and debugging interfaces by watching values in the spreadsheet while the user interface is running.

KEYWORDS: Constraints, Spreadsheets, User Interface Development Tools.

INTRODUCTION

C32 is a tool that helps users construct *constraints*. A ‘‘constraint’’ is a relationship among objects that is declared once and maintained automatically by the system. Typically, a constraint is expressed as an expression (or ‘‘formula’’) that is stored in a slot of an object. The expression is re-evaluated whenever any other values change that are referenced in the formula. Constraints are used to control the graphical objects in many modern user interface toolkits.

C32 uses a spreadsheet model and provides the programmer with the full power of writing constraint code in the underlying programming language. However, it is significantly easier to use, and provides many of the advantages for graphics programming that financial spreadsheets provide for business.

C32 is different from previous spreadsheet systems for user interface construction because it uses a wide array of visual and inferencing techniques so the user does not have to write the entire constraint by hand. In particular:

- C32 automatically generates appropriate references to graphical objects when the user clicks on the object in a user interface window.
- It uses demonstrational techniques to guess which properties of objects should be used,
- It guesses how to parameterize constraints when they are copied from one place to another or generalized into procedures, so abstract and reusable constraints can be constructed *by example*.
- It incorporates graphical techniques to help trace and debug constraints.
- It is integrated with an existing prototype-instance system in which constraints can be inherited.
- It is one of a suite of tools built on top of an existing, successful constraint system, rather than providing the only interface to the constraints, so users can choose other tools when they are more appropriate.

Many systems provide a graphical, direct manipulation technique for specifying some constraints. Unfortunately, such techniques are incapable of expressing every constraint the user may want. C32 provides a convenient and easy-to-use technique for constructing constraints when the graphical techniques are inappropriate. For example, C32 pops up when the user asks for a custom constraint in the Lapidary interactive design tool [8]. C32 can also be used stand-alone when a graphical front end is not available. We have found C32 to be significantly easier to use than constructing the constraints by typing in code.

C32 has been implemented as part of the Garnet system [10]. It is an acronym, and stands for CMU’s Clever and Compelling Contribution to Computer Science in CommonLisp which is Customizable and Characterized by a Complete Coverage of Code and Contains a Cornucopia of Creative Constructs, because it Can Create Complex, Correct Constraints that are Constructed Clearly and

Concretely, and are Communicated using Columns of Cells that are Constantly Calculated so they Change Continuously and Cancel Confusion.

RELATED WORK

Spreadsheets have been used for financial calculations for a long time, starting with VisiCalc in about 1984, and most systems have the same form: an array of cells, with each column labeled with a letter and each row with a number. Some extensions to the spreadsheet idea include using it for logic programming [12] and a technique for adding procedural abstraction [1].

The NoPumpG and NoPumpII systems [15] use spreadsheets to define graphical user interfaces, but they have a number of important differences from C32. The most important is that the NoPump systems provide many different *types* of cells. In C32, all the cells are the same type: slots of objects. To program the user interface, NoPump provided special cell types that reported low-level mouse positions and clock ticks, whereas C32 uses Garnet's high-level Interactor objects [9] to handle behaviors, and slots of Interactors can be specified and viewed in C32 cells, just like any other object. In NoPump, the cells are free floating whereas C32 uses a tabular organization, like a conventional spreadsheet. Thus, all the cells about a particular object are always in one place in C32. There are additional small differences. The cells in NoPump are typed and the formulas use a special language. In C32 the cells can hold any kind of value, and formulas are expressed in a standard language (Common Lisp). Also, NoPump provides few facilities for object referencing, and none for formula generalization.

Constraints have been used by many systems, starting with Sketchpad [13] and Thinglab [3]. Uses of constraints within user interface toolkits include GROW [2], Peridot [7], Apogee [5], and CONSTRAINT [14].

Graphical interfaces to constraints include the ‘‘wiring diagrams’’ in Thinglab [4], the iconic interfaces in Juno and Lapidary [8], and the reference lines in Apogee [6]. The Peridot system automatically infers constraints from example drawings [7]. The wiring diagrams are hard to use for complicated constraints, and the other techniques cannot even handle complex constraints. The spreadsheet interface described here could be used as a *supplement* to these other techniques when they cannot generate the desired relationship.

CONSTRAINT EXPRESSIONS

Objects in Garnet have instance variables or fields, called *slots*. The content of each slot is either a normal value, such as a number or string, or a *formula* that computes the value. References to other objects in formulas use a special form: `(gv other-object slot-name)`, where `gv` stands for ‘‘get value.’’

Because each slot can contain at most one formula, only *one-directional* constraints are supported. We are exploring multi-way constraints for the future, in which case C32 will be changed appropriately.

Through extensive experience with the many projects that have hand-coded Garnet constraints, we have discovered that people have trouble generating correct constraints. Although most constraints in interfaces are very simple, there are a reasonable number of five to ten line code fragments used as constraints. Even the simple constraints can be tricky to enter, since the user must reference the appropriate objects and slots. Also, given a set of constraints that are not working correctly, users have difficulty finding the bugs. C32 was designed to address these problems.

OVERVIEW

C32 can display and allow the user to edit any kind of object and constraint, no matter how they were created: by hand-coding, by using Lapidary (a Garnet interactive design tool), or by using C32.

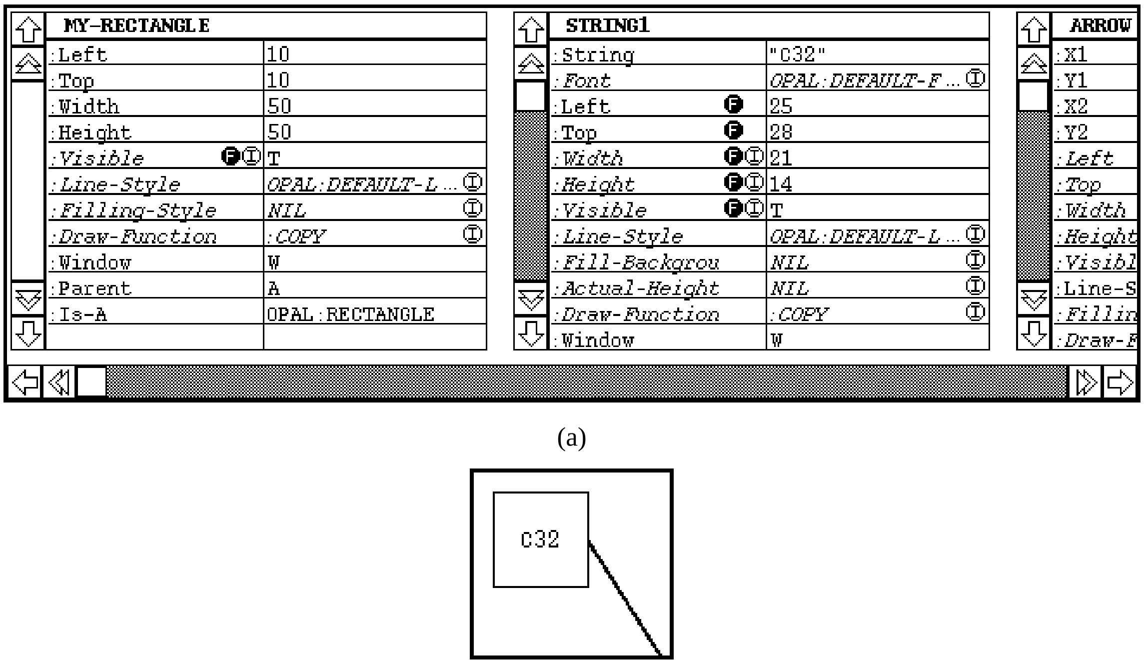
Figure 1 shows a typical instance of C32. Each column contains a separate object. Rows are labeled with the names of the slots, such as `:left, :top, :width, :height, :visible`, etc.[1] Since different objects can have different slots, the slot names are repeated in each column. For example, lines have slots for the endpoints (`:x1, :y1, :x2, :y2`) but rectangles do not. Also, each object's display can be scrolled separately, so each has its own scroll bar. This makes the spreadsheet look somewhat like a multi-pane browser as in Smalltalk.

The spreadsheet cells show the current values of the slots. If a value changes, then the display will be immediately updated. It is important to emphasize that the user interface being constructed will operate normally (albeit a little slower) even while the spreadsheet is displaying objects in that user interface. The underlying constraint mechanism is used internally by the spreadsheet to maintain this connection. Monitoring the values as they change can help the programmer debug objects, and makes the constraints much more ‘‘visible’’ and understandable. If the user edits the value in the spreadsheet cell, the object's slot will be updated.

The F icon by some slots in Figure 1 means that the slot value is computed from a formula. Pressing the mouse on the icon causes the constraint expression to appear in a different window (see Figure 2). The expression itself can be edited by typing or other techniques (discussed later). Note that unlike cells in conventional financial spreadsheets, C32 allows a slot to have a value different from what the formula would calculate. Therefore, the user can edit the value of slots with formulas without affecting whether there is a formula there or not. This can be useful when trying to find the correct value for a slot while debugging. To remove a formula, the user simply deletes the entire string in the formula display window.

One novel feature of the underlying object system is that new slots can be added to objects at any time. Using C32, the user can create a new slot by simply typing a slot name and value into a blank row.

[1] All slot names start with a colon.



(a)

(b)

Figure 1: (a) C32 viewing three objects (b). The scroll bars can be used to see more slots or columns. Changing the window's size will change the number of slots and objects displayed (the number of rows and columns). Field values are clipped if they are too long, but can be scrolled using editing commands. The F icon means that the slot value is computed with a formula. All inherited slots are shown in italics and marked with the I icon. (Inheritance is discussed in a later section.) When a *formula* is inherited the value is shown in a regular font since it is usually different from the prototype's. The inherited icon is also shown next to the formula icon rather than next to the value.

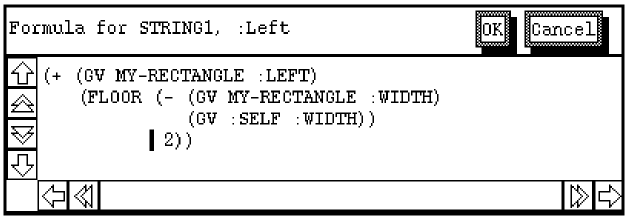


Figure 2: A formula window showing the constraint in the `:left` slot of the `STRING1` object of Figure 1, which centers the string in the rectangle.

To view an object in the spreadsheet, the user can simply type the object's name into the title of a column. Alternatively, the user can select a column, and then point to an object in a graphics window.

As with financial spreadsheets such as Microsoft Excel, we provide a menu of the common functions used in formulas.[2] Another menu contains the graphical commands provided by Garnet, including functions to center objects with respect to other objects and to make their size be proportional. The user can easily add commands to this menu, either written in a conventional way or created from formulas. When functions are inserted from the menus, C32 puts the parentheses in the correct places and leaves the cursor where the arguments to the function go. In this way, C32 can be used like a syntax-directed editor, which has the following advantages:

- the user does not have to deal with the syntax of the language so there will be fewer syntax errors,
- the system will handle the parenthesis matching, which otherwise can be annoying in Lisp, and
- this makes the system accessible for people who do not know Lisp.

GENERATING OBJECT REFERENCES

One of the most interesting aspects of C32 is the way that object references can be specified. As in a financial spreadsheet, the user can point to a slot and have a reference to that slot inserted into the formula at the current point. Figure 3 shows how this can be used.

In Garnet, there are different ways to reference an object in a formula. Unlike other systems such as Peridot and conventional spreadsheets, Garnet allows *indirect* references to objects, where the object to be referenced is stored in a slot of the object that contains the formula. One place this is

[2]There are so many functions in Common Lisp that only the most commonly used are provided in the menu.

used is in composite objects. For example, if a graphical aggregate is composed of a shadow, an outline rectangle, and a label, as shown in Figure 4, then a reference to the left of the shadow from the label would not name the shadow directly. Instead, the reference would be:

```
(gv-indirect :parent :shadow :left)
```

These indirect references make it much more efficient to create copies and instances of aggregates, since it is not necessary to search through all the formulas and change the references to refer to the new objects. When the formula and the slot being referenced are part of the same aggregate structure, then an indirect reference like the one described above must be generated. If the objects are totally distinct, then a direct reference can be used. C32 searches the object hierarchy to decide which is appropriate.

USE OF INFERENCING

It is sometimes not convenient to read an object into a spreadsheet column just to generate a reference to it. Therefore, a command will cause the system to go into a mode where a graphical object in any Garnet window can be selected and a reference to it placed into the current formula. However, selecting a graphical object does not specify which *slot* of the object should be referenced. In one mode, the user must type this directly or select a slot from a menu. However, there are two inferencing modes that try to guess the slot from the user's actions. One uses the current relationship of the two graphical objects to guess the desired constraint, much like Peridot [7]. For example, if the slot being edited is `:left` and the object seems to be centered horizontally with respect to the selected object, then C32 generates a centering constraint.

The other mode ignores the current positions of the objects, but looks at the slot being filled and where the mouse is pressed in the selected object. For example, if the slot is `:left`, and the mouse is pressed at the right of an object, then the reference will be to the right of the object. For the `:width` slot, however, the same press would generate a reference to the width of the object. Unlike Peridot, C32 does not try to confirm any of the inferences, but rather simply inserts the text into the formula. If the guess is incorrect, it is easy for the user to delete the text and type the correction.

Once a complex formula is created, it will often be needed in a slightly different form for a different slot or a different object. As an example, suppose the user has constructed a constraint that centers an object horizontally with respect to two other objects (see Figure 5). Now, suppose the programmer wants to center the object vertically also. The formula could be copied to the `:top` slot, but all the slots references need to be changed (`:left` to `:top` and `:width` to `:height`). Therefore, when a formula is copied, C32 tries to guess whether some slot names should be changed. This uses a few straightforward rules based on the slot names of the source and destination slots. If it

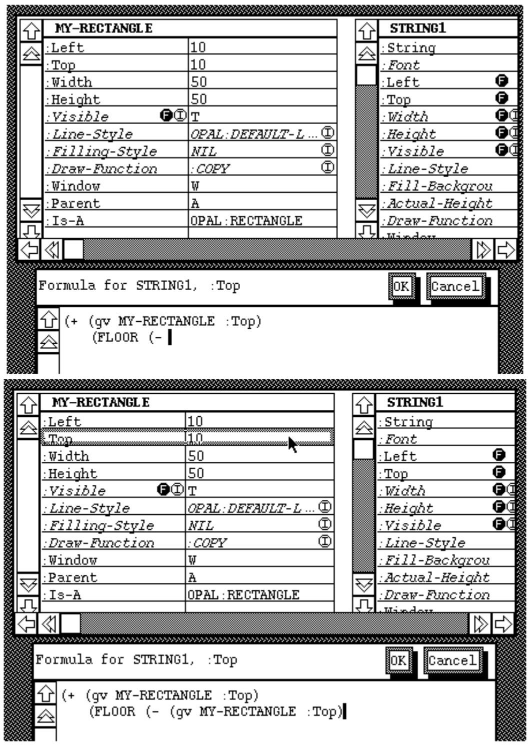


Figure 3: The spreadsheet before and after the user has selected the `:top` cell of `MY-RECTANGLE` to be inserted into the formula.

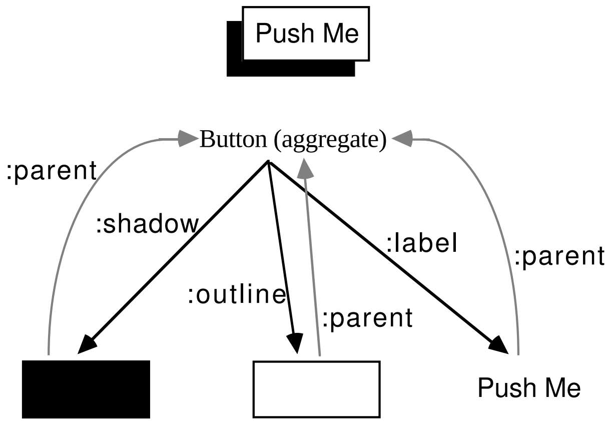
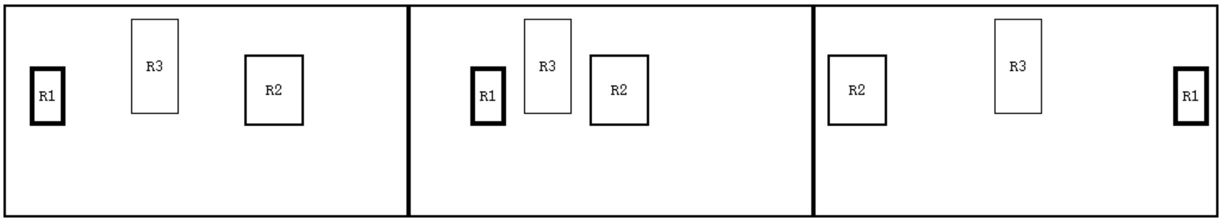


Figure 4: A graphical button and its aggregate hierarchy. References from one object to another use paths through the hierarchy. Objects that are part of the button have programmer-assigned names, like `:shadow` and `:outline`, and references to the button from its parts uses the standard `:parent` slot. In a slot of the shadow, a reference to the `:width` of the text label would be `(gv-indirect :parent :label :width)`.



(a)

```
(formula '(floor (+ (gv r1 :left)   ;floor does int divide
                    (gv r1 :width)
                    (gv r2 :left)
                       ;get this object's width.
                    (- (gv-indirect :width)))
                 2))
```

(b)

Figure 5: (a) Rectangle `R3` is centered horizontally between `R1` and `R2` using the formula shown in (b) so `R3` moves automatically when `R1` and/or `R2` moves.

appears that slot names should be changed, the user is queried with a dialog box, and if the answer is OK, then the formula is modified automatically.[3]

AUTOMATIC GENERALIZATION

Another possibility is that the references in the formula should be *generalized* into variables. C32 therefore provides a command that will change the entire formula into a function that takes the objects and/or slots as parameters. This process is controlled by a dialog box. As an example, generalizing the formula of Figure 5 creates the dialog box of Figure 6-a, and the code would be changed as shown in Figure 6-b. The new `Center-Between-X` function can now be used in other formulas. It will also be added to the C32 graphics functions menu, so it can be easily retrieved later.

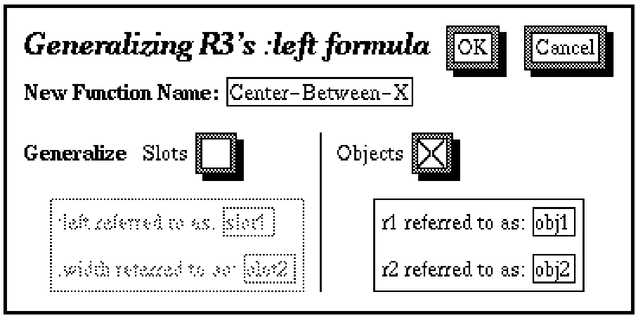
The intelligent copying and generalizing discussed here helps the user generate correct constraints *by example*. Without these aids, it is quite common to forget to change one or more of the references when formulas are copied. Generalizing also helps the programmer decrease the size of the code by promoting the reuse of existing formulas. Future research will investigate further ways to use generalization.

TRACING AND DEBUGGING

Experience with Garnet and all other constraint-based systems shows that people have a difficult time debugging their formulas. The primary problem is that constraints are not local because values in one object can affect values in many different objects. Therefore, C32 provides a set of tracing and debugging tools.

The most straightforward method is to simply use the spreadsheet to exercise and monitor the user interface in action. Often, seeing the current values of all the slots is sufficient to determine the problem. To help relate the cells and objects, commands are provided to blink the object associated with a cell, or the cells for an object.

Other commands display arrows in the spreadsheet to show which cells are used in the computation of the current cell (see Figure 7-a) or the cells that use the value of this cell (Figure 7-b). We are exploring additional graphical constraint debugging tools.



(a)

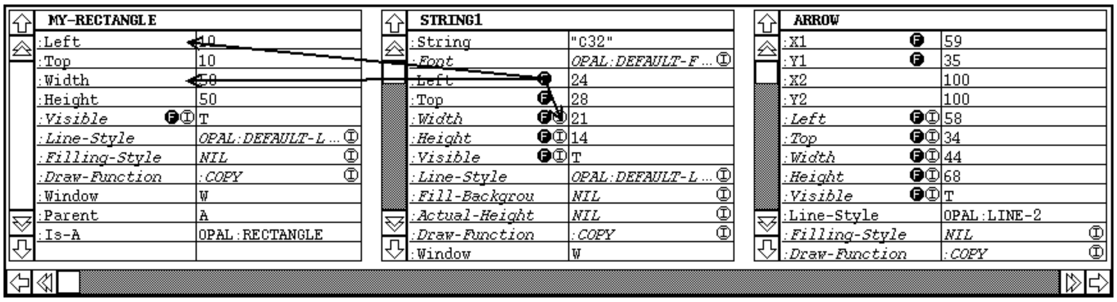
```
(defun Center-Between-X (obj1 obj2)
  (floor (+ (gv obj1 :left)
            (gv obj1 :width)
            (gv obj2 :left)
            (- (gv-indirect :width)))
         2))

(formula '(Center-Between-X r1 r2))
```

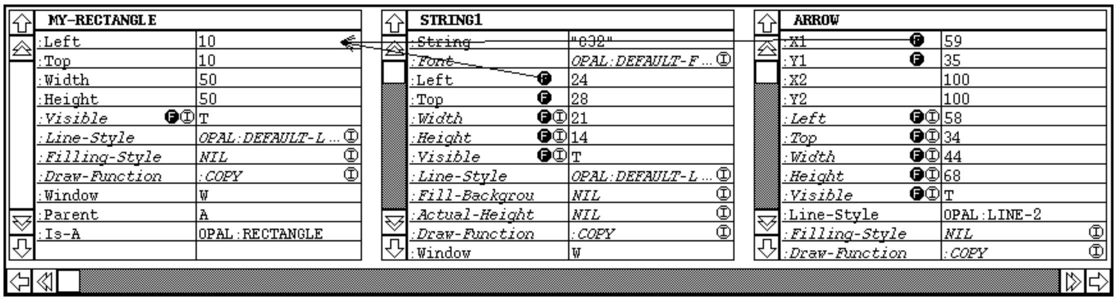
(b)

Figure 6: (a) The dialog box for generalizing from a formula. All the values shown are the defaults provided by C32. After the user hits `OK`, the formula of Figure 5-b is converted automatically into the function shown here in (b).

[3] Since this is a more radical change than the inferred slots discussed in the previous section, it seems prudent to require confirmation.



(a)



(b)

Figure 7: C32 will display arrows to show which slots the current one depends on or which slots depend on the current cell. (a) shows that the `:left` of `STRING1` depends on the `:left` of the rectangle, and the `:widths` of the rectangle and itself (since it is centered). (b) shows that the `:left` of the rectangle is used by the `:left` of the string and `:x1` of the line. In both views, the arrows point to the slot being used.

INHERITANCE

Garnet uses a prototype-instance model instead of the conventional class-instance model. This means that any object can be used as a prototype for a new set of objects; there is no distinction between classes and instances. One result of this is that each instance decides which slots to inherit and which to override. For example, many graphical objects do not have a `:filling-style` slot, but rather inherit this value. Inherited slots are shown in italics with the Ⓘ icon next to their value (see Figure 1).

Inheritance in Garnet is determined dynamically. This means that setting the value of a slot which used to be inherited, changes the slot to be local with the new value. C32 shows this by removing the inherited icon after a slot is edited. When a local slot is deleted from an object in Garnet, the system looks in the prototype to see if that same slot exists there, and if so, the slot becomes inherited. Therefore, if a slot is deleted in C32 but there is a value that can be inherited, C32 will change the display to show the inherited value. This makes it clear to the user that although the local slot is removed from the instance, there is still a legal value for it.

Formulas can also be inherited. In this case, there is often a different current value for the slot, even though the formula is the same as the prototype's. For example, the `:width` slot of a text object usually contains a formula that computes the width based on the object's font and string value. Most text objects inherit this formula, but still have a different current value because their string and font values are different. Therefore, if a formula is inherited, the inherited marker is shown next to the formula icon in the spreadsheet and the value part is not shown in italics.

INTEGRATION WITH OTHER TOOLS

There are many different mechanisms and tools in the Garnet system. Therefore, unlike previous spreadsheet systems such as NoPump, C32 does not have to address every aspect of user interface design.

When the programmer wants graphics in Garnet to respond to the mouse or keyboard, an Interactor object [9, 11] is attached to the graphics to handle the input events. There is a standard protocol that the Interactors use to query and modify the graphics. Since Interactors are objects, they can also be read into C32. Unlike graphical objects, however, Interactors are not visible so they cannot be pointed at. Therefore, there are commands to display a menu of all the Interactors, of all those that affect a particular graphic object, or all those that respond to a particular input event.

C32 can also be used with the Lapidary interactive user interface design tool [8]. Lapidary allows the designer to draw pictures of what the user interface should look like and then demonstrate how it should act. Although Lapidary provides an iconic interface to many common constraints, C32 can be used when more complex or custom constraints are necessary.

STATUS AND FUTURE WORK

The spreadsheet described here is mostly working, and we expect to release it to Garnet users within the next few months. Their feedback will be valuable in deciding what features to add and modify. We expect to explore:

- Other ways to use demonstration to create formulas.
- More clever generalizations from existing formulas.
- Better connection with Interactors. There is a well-defined protocol between Interactors and graphic objects that serve as feedback objects. The spreadsheet could set the appropriate fields of the graphic object automatically if the object was placed in a slot of the Interactor, as is done by Lapidary [8].
- Ways to use C32 to create objects from scratch, so C32 can be used as an interface builder. Once objects have been created in memory, Garnet already contains a built-in mechanism that will write them to a file so they can be used by real applications.

CONCLUSION

The C32 spreadsheet contains a number of novel features, including the use of demonstrational techniques to generate object references, automatic generalization of formulas, and graphical tracing and debugging. These make it easier to use than previous spreadsheet-based graphical tools. C32 enhances the Garnet user interface development environment by providing an appropriate mechanism for specifying complex, custom constraints, which occur frequently in user interface software. C32 has demonstrated that a spreadsheet tool can be a valuable addition to an existing constraint-based system, and that it is possible to get totally carried away in acronym building.

ACKNOWLEDGEMENTS

Brad Vander Zanden contributed to the design of C32. An earlier version of C32 (then called C29) was implemented by Andrew Mickish. The Garnet system as a whole has been developed by a large number of people. For help with this paper, I want to thank Brad Vander Zanden, Bernita Myers and the referees.

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Additional support for Garnet was supplied by Siemens, Apple Computer, Inc. and General Electric.

REFERENCES

1. Allen L. Ambler. Forms: Expanding the Visualnes of Sheet Languages. 1987 Workshop on Visual Languages, Visual Language'87, Linkoping, Sweden, Aug., 1987, pp. 105-117.

2. Paul Barth. "An Object-Oriented Approach to Graphical Interfaces". *ACM Transactions on Graphics 5*, 2 (April 1986), 142-172.

3. Alan Borning. Thinglab--A Constraint-Oriented Simulation Laboratory. Tech. Rept. SSL-79-3, Xerox Palo Alto Research Center, July, 1979.

4. Alan Borning. Defining Constraints Graphically. Human Factors in Computing Systems, Proceedings SIGCHI'86, Boston, MA, April, 1986, pp. 137-143.

5. Tyson R. Henry and Scott E. Hudson. Using Active Data in a UIMS. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'88, Banff, Alberta, Canada, Oct., 1988, pp. 167-178.

6. Scott E. Hudson. Graphical Specification of Flexible User Interface Displays. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'89, Williamsburg, VA, Nov., 1989, pp. 105-114.

7. Brad A. Myers. *Creating User Interfaces by Demonstration.* Academic Press, Boston, 1988.

8. Brad A. Myers, Brad Vander Zanden, and Roger B. Dannenberg. Creating Graphical Interactive Application Objects by Demonstration. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'89, Williamsburg, VA, Nov., 1989, pp. 95-104.

9. Brad A. Myers. Encapsulating Interactive Behaviors. Human Factors in Computing Systems, Proceedings SIGCHI'89, Austin, TX, April, 1989, pp. 319-324.

10. Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. "Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces". *IEEE Computer 23*, 11 (Nov. 1990), 71-85.

11. Brad A. Myers. "A New Model for Handling Input". *ACM Transactions on Information Systems 8*, 3 (July 1990), 289-320.

12. Michael Spenke and Christian Beilken. A Spreadsheet Interface for Logic Programming. Human Factors in Computing Systems, Proceedings SIGCHI'89, Austin, TX, April, 1989, pp. 75-80.

13. Ivan E. Sutherland. SketchPad: A Man-Machine Graphical Communication System. AFIPS Spring Joint Computer Conference, 1963, pp. 329-346.

14. Brad T. Vander Zanden. Constraint Grammars--A New Model for Specifying Graphical Applications. Human Factors in Computing Systems, Proceedings SIGCHI'89, Austin, TX, April, 1989, pp. 325-330.

15. Nicholas Wilde and Clayton Lewis. Spreadsheet-based Interactive Graphics: from Prototype to Tool. Human Factors in Computing Systems, Proceedings SIGCHI'90, Seattle, WA, April, 1990, pp. 153-159.