Journal of Visual Languages and Computing (1992) 3, 25–47

GVL: Visual Specification of Graphical Output

JAMES R. CORDY* and T. C. NICHOLAS GRAHAM

**Department of Computing and Information Science, Queen's University at Kingston, Kingston, Canada K7L 3N6 and GMD Forschungsstelle an der Universität Karlsruhe, Vincenz-Prießnitz-Str. 1, D-7500 Karlsruhe 1, Germany*

Received 17 June 1991 and accepted 24 October 1991

The conceptual view model of output is based on the complete separation of the output specification of a program from the program itself, and the use of implicit synchronization to allow the data state of the program to be continuously mapped to a display view. An output specification language called GVL is used to specify the mapping from the program's data state to the display. GVL is a functional language explicitly designed for specifying output. Building from a small number of basic primitives, it provides sufficient power to describe complex graphical output. Examples shown in the paper include GVL specifications for linked list diagrams, bar charts and an address card file. In keeping with its intended application, GVL is also a visual language in which the user draws output specifications directly on the display. It is shown how problems often associated with imperative graphical languages are avoided by using the functional paradigm. A prototype implementation of GVL was used to produce all examples of graphical output in the paper.

1. Introduction

THE MODEL of input/output used by most modern programming languages is based on streams. A stream is a one-dimensional I/O channel: input characters are taken from the front of the input stream and output characters are appended to the end of the output stream. Output occurs only from specific points in the program where output statements have been inserted. This stream model most easily supports the glass teletype model of user interaction, where input and output take place on the bottom line of the screen, and earlier interactions are scrolled up onto the remaining screen lines. This glass teletype interaction leads to a prompt-and-respond program interface, where the program prompts for the requried input in some order, and the user is obliged to respond in that order.

Sufficient diversity in interface and hardware design has occurred that this view is no longer adequately representative [1]. For example, sophisticated user interfaces are two-dimensional, using the full display for interaction. Often the user is permitted to direct the interaction, filling in inputs in whatever order is convenient. This style of direct manipulation interface is difficult to implement when the programming language provides only steam-level communication.

In recent years considerable work has gone into developing high level constructs in programming languages to support better abstract data types. In modern application programs, between 29 and 88% of the program code is required to implement the user

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, and by the Information Technology Research Centre.

1045-926X/92/010025 + 23 $03.00/0

interface [2]. It is therefore appropriate to develop similar high level abstractions for program input and output.

This paper describes GVL (Graphical View Language), a graphical, functional language used to specify output. GVL is used to specify conceptual views of output, which map the data state of an application program to a display view. Following a brief overview of the conceptual view model, the paper describes GVL, the language on which the model is based. (Earlier papers describe the conceptual view model in more detail [3, 4].) Weasel, a prototype implementation of the conceptual view model, was used to produce all of the output shown in this paper.

2. The Conceptual View Model

The conceptual view model is based on a separation of a program's output from the program itself. Figure 1 shows a typical program organization under the conceptual view model. The program consists of a set of modules, encoded in the language of choice of the application programmer (e.g. C, Turing, Ada, etc.) The program contains no output statements, therefore containing only data structures and algorithms to manipulate the data structures.

A separate output specification maps the contents of a data structure to a display view. As the data structure is modified, the display view is automatically updated. Conceptual views can be thought of as a data probe, the software equivalent of a logic probe. Each probe continuously senses the data state of the program and maps it to a display view. The display view is an abstraction of the data structure, representing some facet of the data structure that is of interest to the programmer; the name conceptual view comes from this idea of abstraction.

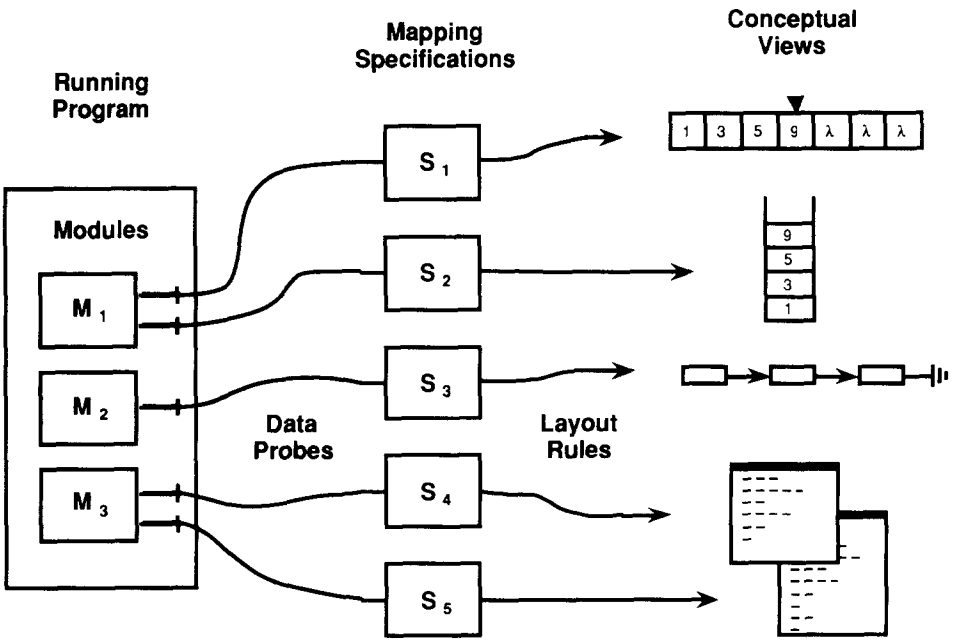

Figure 1. Typical program organization under the conceptual view model

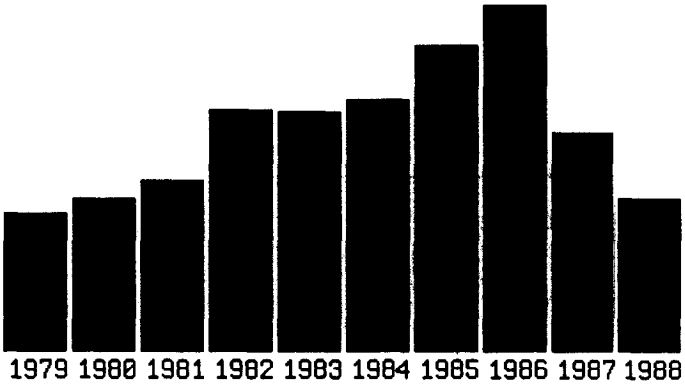

Figure 2. Bar graph output generated as a conceptual view of a list of real numbers

A data structure can be mapped to multiple different output views. For example, in Figure 1, the specification S1 maps the data structure implemented in module M1 to an arry diagram with a cursor, while specification S2 shows the same data structure as a stack.

As the data structure is modified throughout the execution of the program, the display is implicitly updated. These updates are based on the invariant assertion of the

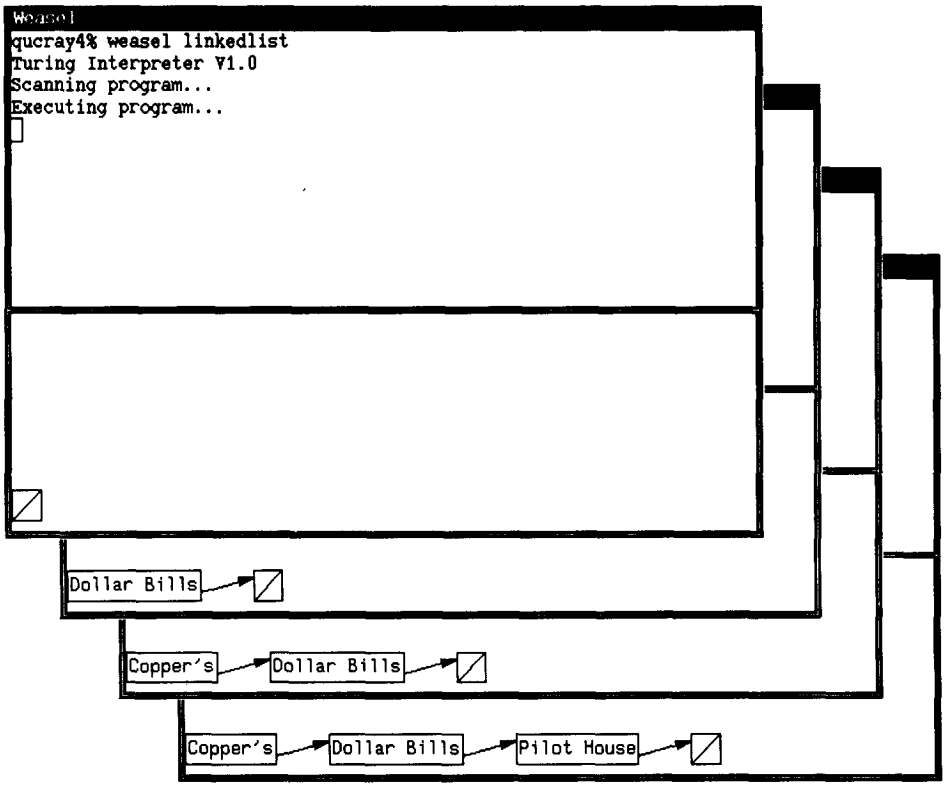

Figure 3. A series of snapshots over time of a linked list data structure as it is being constructed

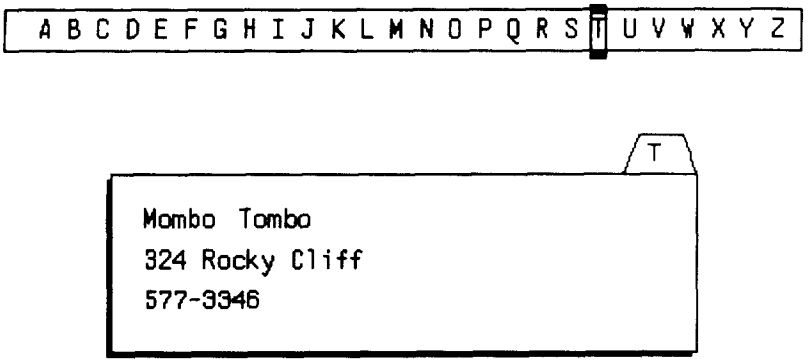

Figure 4. A card file name and address interface implemented as a conceptual view

module being displayed: when the invariant is false (i.e. an update is taking place), it is illegal to update the display. When the invariant is true, the module's data structure is guaranteed to be in a consistent state, and the display can be updated. Therefore, it is sufficient to update the display every time the module's invariant changes from false to true, which corresponds to the module being exited.

Conceptual view specifications are expressed in two parts. First, a specification written in GVL expresses how the current state of the data structure is to be mapped to a set of display primitives such as boxes, lines and text. The mapping specification does not necessarily constrain the location or sizes of these primitives. Then, a set of layout rules are applied to this form to determine the actual display. Layout rules determine any unconstrained sizes and positions, and resolve how to fit large displays when the physical display device is too small.

Figures 2, 3 and 4 show examples of conceptual views generated by the Weasel protoype. Figure 2 shows a bar graph as the output of a list of real numbers. Figure 3 is a series of snapshots over time as a linked list is constructed. Figure 4 shows a snapshot of an interface implementing a card file address book. This paper describes how GVL can be used to specify these conceptual views.

3. A Simple Example

This section gives a brief example of a GVL output specification. This example is only an overview of how such a specification is constructed; more detailed examples are given later in the paper.

Consider that a programmer wishes to implement a database for names and addresses. The user interface is modeled after a card file index, where the user can move back and forth within the card file, and the current card is displayed on the screen. The base program will contain the data structure to represent the card file, and will encode the algorithms for traversing the file. Separately, a conceptual view is to be specified to map the card file data structure to a screen view of the current card.

Figure 5 shows the GVL function to map from the card file data structure to a display view of the card. The programmer has drawn the card as he/she intends it to be seen. The card frame itself is drawn as a box, with a filled border to create the illusion of a shadow. Within the box, applications of additional display functions are

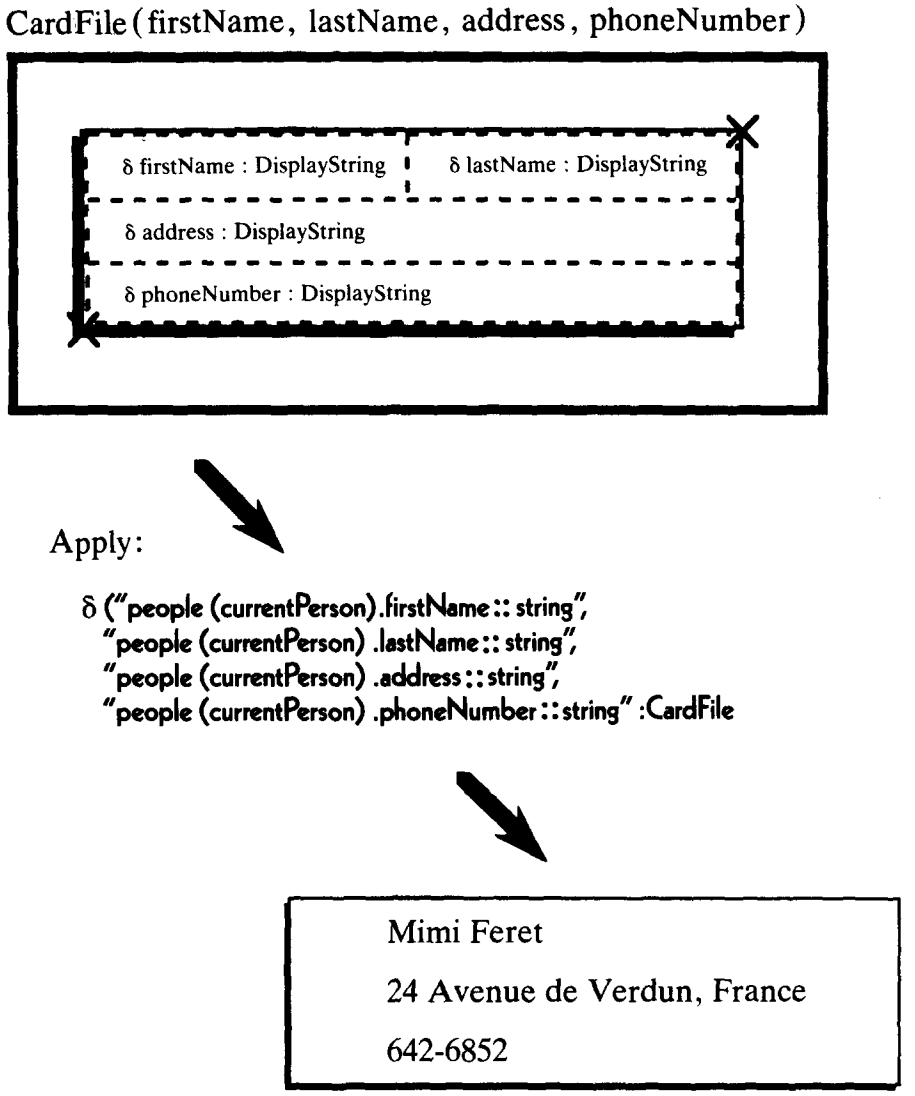

Figure 5. GVL function to implement part of the card file output shown in Figure 4, an application of this function, and the resulting output

used to indicate where the name, address and phone number information is to be placed on the card. Each application is of the form $\delta p:f$, where p is one of the parameters to the CardFile function, and f is the name of the display function to be applied. For example, δ ***firstName:DisplayString*** indicates the first name is to appear in the upper-left corner of the card. (**DisplayString** is a predefined function that displays a character string.) Traditional notation for $\delta p:f$ might be $f(p)$. The δ notation is used instead since it is easily read as 'display p as an f', which reflects the intuitive semantics. For example, δ ***rest:LinkedList*** means 'display ***rest*** as a ***LinkedList***'.

```
type person:
  record
    lastName:string
    firstName:string
    address:string
    phoneNumber:string
  end record
var people:
  array nullPerson..numberOfPeople of person
var currentPerson:=nullPerson
```

Figure 6. A Turing language data structure to implement the card file database

The **CardFile** function has four parameters, ***firstName***, ***lastName***, ***address*** and ***phoneNumber***. These four functions are to be bound to source expressions written in the application programming language that refer to the data structure to be viewed. Figure 6 shows a data structure encoded in the Turing language [5] that could be used to implement the card file. The data structure consists of an arry of records, one record per card. The ***currentPerson*** variable indicates which card is currently selected.

To display the card file, the programmer can use the following application of the ***CardFile*** function

$$\delta\ (\text{"}people\ (currentPerson).firstName::string\text{"},\ \text{"}people\ (currentPerson).lastName::string\text{"},\ \text{"}people\ (currentPerson).address::string\text{"},\ \text{"}people\ (currentPerson).phoneNumber::string\text{"}):CardFile$$

This means, for example, that whenever the ***FirstName*** parameter is to be displayed, the expression 'people (currentPerson).firstName' is evaluated to produce a character string to be displayed on the output device.

The remainder of this paper describes GVL, a graphical functional language used to specify conceptual view mappings.

3. GVL: a Graphical, Functional Language

A conceptual view mapping m is a mapping from the current state of a data structure to a display view: i.e.

$$m: programState \times layoutRuleSet \rightarrow display$$

where a ***programState*** is the state of the application program at some instance, and a ***layoutRuleSet*** is a set of rules to guide the exact placement and sizing of the primitive elements of the display. We have found it convenient to describe these view specifications graphically using GVL.

GVL functions take a list of parameters and yield a conceptual view mapping. In particular, a GVL function is a function f such that

$$f: parameterList \rightarrow (programState \times layoutRuleSet \rightarrow display)$$

where each parameter in a parameter list can be a source expression encoded in the application programming language, a list of such expressions or another GVL function.

The application of a GVL function results in a conceptual view, which in turn can be applied to the program state to produce a display. Figure 5 showed the application

of the *CardFile* function to a list of expressions to produce a conceptual view that maintains a display of the current state of a card file database.

4. Why a Language?

A notation used to express conceptual view mappings requires at least some of the features normally associated with a programming language. It is therefore desirable to define the notation as a language, and to attribute to it a well-defined syntax and semantics. To be a programming language, features equivalent to abstraction, repetition and selection must be provided. The following paragraphs show why these features are required in the conceptual view specification language.

4.1. Abstraction

In specifying output, many displays are used over and over again. For example, a programmer would wish to re-use the same style of menus and dialogue boxes from program to program. Similarly, when developing a complicated interface, a programmer would wish to be able to develop different parts of the interface in isolation, and combine them later. Shaw *et al.* [1] discuss the need for hierarchies of predefined abstractions to aid in the development of program interactions. GVL provides function definitions as an abstraction mechanism.

4.2. Repetition

A bar graph display (Figure 2) requires a sequence of bars to be drawn, each one beside the previous. A display of a linked list (Figure 3) requires a sequence of list elements to be drawn. Each of these examples shows the necessity of including a mechanism for repeating the same action an arbitrary number of times. GVL allows the use of recursion for repetition.

4.3. Selection

In the example of the bar graph or linked list above, it is necessary to be able to test when to stop repeating. In other cases, a display may be tailored to be of a different form depending on the user's request. These examples show the need to be able to specify some form of conditions in output specifications; GVL provides a *cond* primitive for selection.

The need for mechanisms for abstraction, repetition and selection imply that the mapping specification language should be a full programming language.

5. Why Graphical?

The traditional approach has been to use a linear notation (i.e. text) to specify graphical output. Rather than a user having to define a coordinate space and specify the locations and sizes of objects in terms of numerical coordinates, a graphical notation allows direct manipulation of a two-dimensional display to specify where the various elements of a display are to be located and what size they are meant to be. To specify graphical output, a visual language is less clumsy, more direct and therefore less prone to error.

6. Why Functional?

Given the decision to use a visual language, the functional paradigm presents some strong advantages. In GVL, a small, powerful language was sufficient to achieve the goals of combining flexibility with convenience. As examples will show, the high level nature of the functional language simplifies specifications over imperative styles. For example, infinite recursion is used in some GVL functions to simplify the expression of the required output.

The functional style is appropriate to a graphical representation. Earlier graphical languages have been mainly general-purpose languages, as opposed to special-purpose output languages, and have tended to follow the imperative paradigm. Imperative graphical languages tend to look like flow charts, and inherit their tangled control flow. Graphically representing variables can be a problem; the Pict system [6], for example, uses colours to differentiate between variables. This not only loses the mnemonic nature of the variable name, but also limits the number of variables to four (i.e. the number of colours supported by the display). In a functional style these difficulties are avoided. Since the only control construct is function application, spaghetti coding is avoided. Functional languages do not have variables or assignment, eliminating the problems of how to represent state manipulations.

A functional language also supports the building of programs from separate components. When programming with a graphical language, it is common to wish to draw two different programs and then combine them into one. In a functional style there is no danger that one of the displays will have a side effect that causes it to interfere with the other.

Finally, there is a wide class of known optimizations for functional languages. By restricting the language to being purely functional, optimizations such as tail-recursion elimination, inlining of functions and memoization (removal of duplicate calls) are available [7].

In summary, the need for abstraction, selection and repetition imply that we should design a language for output specification. Graphical specifications are more convenient than textual ones for this purpose. Finally, a functional language appears to be appropriate for graphical programming.

7. Language Constructs

Figure 7 shows the GVL primitives. In general, primitives must appear exactly as they are drawn: if a line is 2 cm long in the specification, it must be 2 cm long in the displayed output. This constraint on the location and size of a primitive is expressed by placing an '×' symbol on a vertex to be constrained. This symbol indicates the vertex is to be drawn exactly where it is shown in the current definition. If a coordinate is not constrained with an '×' symbol, its location is chosen according to a set of layout rules.

GVL uses the concept of *a coordinate space* in a way analogous to the concept of scope in traditional programming languages. A coordinate space is simply an area of display space. A display whose position is unconstrained is restricted to being displayed within the coordinate space in which it is specified. Three language constructs introduce a new coordinate space: function definitions, boxes and the *cond* function. (Note that since boxes can be nested, so can coordinate spaces.)

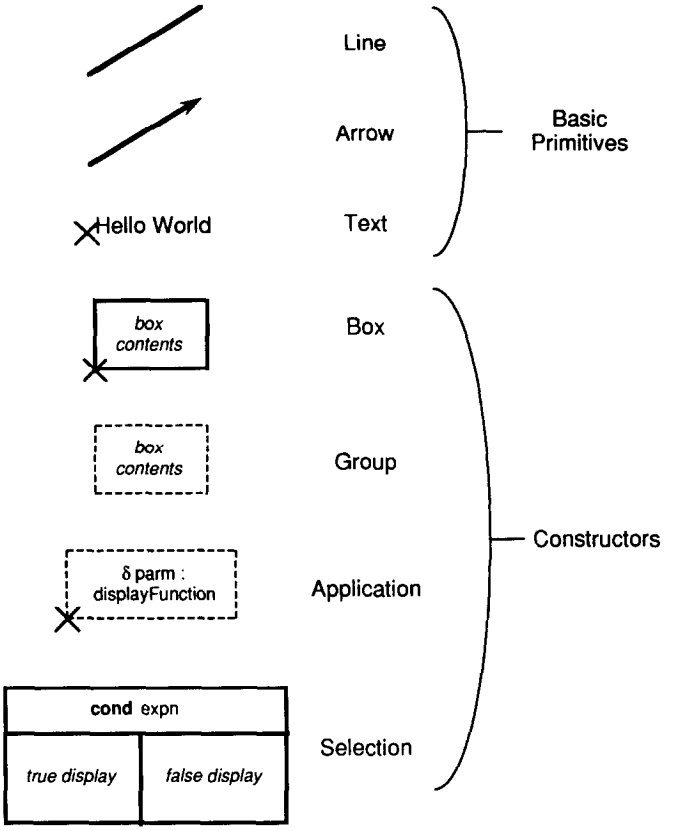

Figure 7. GVL language primitives

8. Basic Primitives

There are three basic primitives: *line*, *arrow* and *text*. Either end of a line or arrow may be constrained with an '×' symbol. If both ends are constrained, the line or arrow is drawn exactly as it appears on the display. If either end is unconstrained, the layout rules are free to place that end-point anywhere on the display. A completely unconstrained line or arrow may be drawn as an arc or curve. Various style attributes may be associated with a line or arrow. In the current implementation, lines may be solid, dashed, dotted or invisible. It is possible to imagine many other attributes (such as colour or thickness) that could be given to lines and arrows.

Text is displayed exactly as it is presented on the screen. If any vertex of the text is constrained, then all four vertices will be constrained. If the location of text is unconstrained, it may be drawn anywhere within the current coordinate space.

9. Boxes

A box primitive evaluates the contents of the box, and evaluates to a box surrounding this display. If the box's vertices are constrained, the resulting box always has the

same dimensions; if the upper-right corner of the box is unconstrained, then the box is sized to fit the contents. A box has a style attribute associated with it. While many other useful styles could be envisioned, the current set includes outlined, invisible and filled boxes.

10. Conditions

The *cond* primitive takes three arguments: a condition expression and two displays. The condition is evaluated. If it is true, then the result of the cond is the display on the left; if the condition is false then the result is the display on the right.

11. Functions

Figure 5 shows an example of a GVL display function definition. This example shows an abstract function, i.e. one that has not yet been bound to any particular data structure. A function definition consists of a parameter list and a body. A parameter may be an expression, a list of expressions or another display function. GVL functions can be higher order (i.e. they can take functional parameters) and polymorphic (i.e. the type of parameters is not bound in an abstract function.) The function body contains any sequence of primitives.

Display function application is specified with the 'δ' operator. In general, application is of the form

$$\delta p : f$$

meaning that the GVL function f is to be applied to the parameter p, and the result is to be merged into the current coordinate space. The result of applying f is a display, which will may contain constrained and unconstrained primitives. The constrained primitives are placed within the dashed box surrounding the application. Any unconstrained primitives remain unconstrained in the current coordinate space, and can therefore be placed anywhere in the coordinate space.

When evaluating the body of a GVL function, it is guaranteed that if the same display is generated twice in a coordinate space, and the location of at least one of them is unconstrained, the two displays will be drawn in the same location. Examples following this section show how useful this property is in simplifying specifications.

Sometimes it is inconvenient to apply a GVL function directly to its arguments. At other times, it may be known in advance that a function is to be applied to some class of similar data structures. In these cases, it is helpful to be able to specialize a display function to a particular data structure or class of data structures before the display function is applied.

This specialization process consists of binding each parameter in the function to some value; each value may be a textual expression from the application program, a list of expressions or the name of a display function. Expressions may themselves be parameterized. The binding creates a new specialized display function, with a new list of parameters. Bound functions may themselves be specialized, allowing the creation of a hierarchy of bound specifications [4]. A later section of this paper discusses the details of the binding process.

12. An Example

An example that captures many of the features of the GVL language is the ***LinkedList*** function (Figure 8). Figure 3 showed an example of output when this function is applied to a list of strings. The function takes four parameters. The ***first*** parameter is the value of the first item of the list; the ***rest*** parameter is all list elements that follow the first. The ***end*** parameter is a condition which specifies when there are no more elements in the list. Finally, the ***DisplayElementType*** parameter specifies what function is to be used to display the items themselves.

In the main part of the ***LinkedList*** function, the first element of the list is drawn in a box. An arrow is then drawn to a display of the remaining element of the list. These remaining elements are drawn by recursively applying the ***LinkedList*** function to the ***rest*** parameter. The first element of the list is displayed using the ***DisplayElementType*** function, which is a parameter to the ***LinkedList*** function. In this way, the same ***LinkedList*** function can be used to display a linked list of strings, a linked list of integers or even a linked list of linked lists.

The display of the ***first*** element is enclosed in a box. The lower-left corner of the box is constrained with an '×' symbol, meaning that the box is to be located in the lower-left corner of the display. The upper-right corner is not constrained, meaning that the box is to be sized to fit the display of the first element. The display of the ***rest*** of the list is not constrained at all, meaning that the subsequent elements can be drawn anywhere in the current coordinate space.

The display is surrounded by an application of the ***cond*** function, which states that if the end of the list is reached (i.e. the ***end*** parameter evaluates to true) then a terminator box is to be drawn (the display on the left), otherwise the main display involving the recursive display of the list is to be used.

The ***LinkedList*** function can potentially be infinitely recursive. If the linked list is malformed such that it has a loop in it, then the ***end*** condition will never evaluate to ***true***. This case is solved by the guarantee that if the same display is generated twice in the same coordinate space, and if the position of at least one of them is unconstrained, the pair will be drawn in the same location. In the case of a list with a loop, at some point in the recursion, the result will consist only of elements that have already been displayed. Since the location of the recursively generated items is not constrained, the repeated items can be drawn in the same location as the items generated earlier. This means that once all the items have been drawn once, the recursion can be terminated.

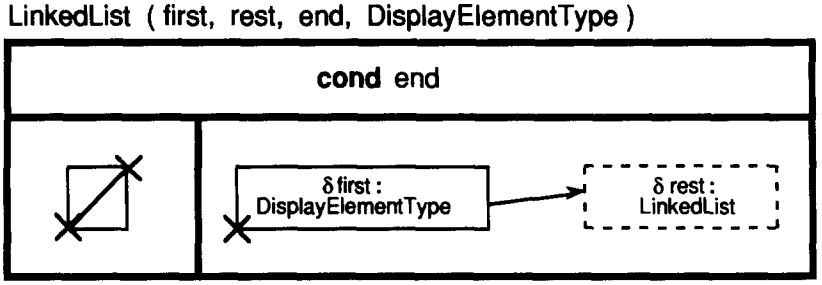

Figure 8. GVL function to draw the linked lists shown in Figures 3 and 9

(This is consistent with the traditional definition of recursion as being the fixed point of an infinite sequence of recursive applications [8].) The implementation of infinite recursion detection (discussed in Nicholas Graham [4]) is similar to the memoization optimization performed in many other functional language implementations [7]. Figure 9 shows the result of applying the *LinkedList* function to a linked list containing a cycle.

13. Binding

Figure 10 shows a Turing language data structure implementing a linked list. The data structure is based on an array of text lines. In order to apply the **LinkedList** function to this data structure, it is convenient to first bind the function to a form specialized to the data structure. This binding is expressed using the following notation

```
LinkedListTextLine (pos) is LinkedList where
   first = "lines (!!pos).text::string",
   rest = "lines (!!pos).nextLine::lineReference",
   end = "!!pos = nilLine::boolean",
   DisplayItemType = DisplayString
specializing LinkedList to LinkedListTextLine
```

This says that a new function, ***LinkedListTextLine***, is to be created. The new function takes one parameter (***pos***), and is defined as being ***LinkedList*** where the first three parameters are bound to expressions referring to the data structure, and the fourth parameter is bound to the ***DisplayString*** function.

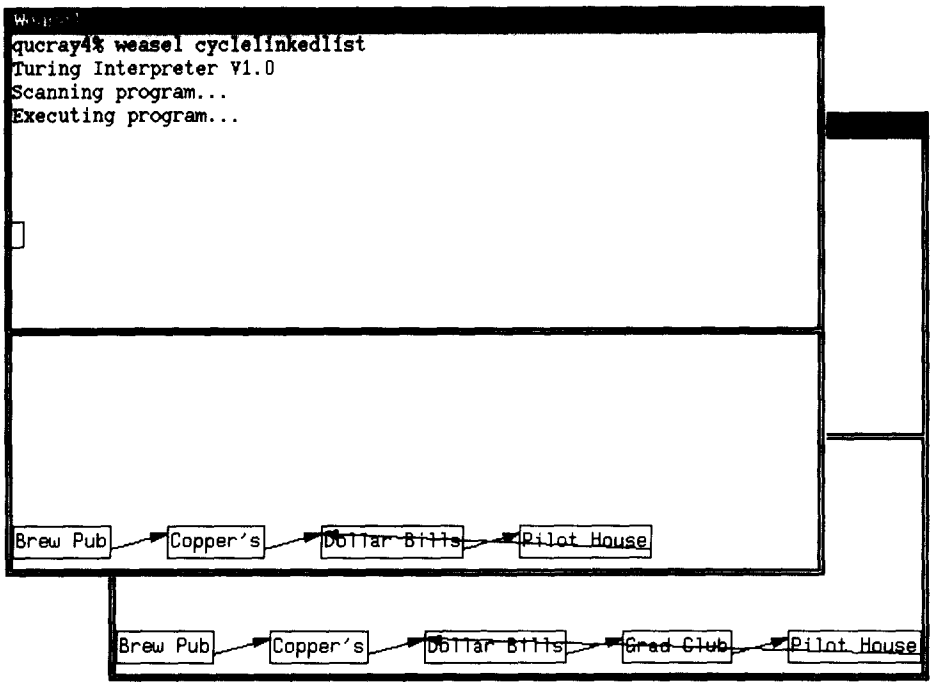

Figure 9. Result of applying the ***LinkedList*** GVL function to a broken (cyclical) linked list

```
const nilLine := 0
const maxLines := 1000

type lineReference : nilLine .. maxLines

var lines : array lineReference of
  record
    text : string
    nextLine : lineReference
  end record
var rootLine : lineReference := nilLine
```

Figure 10. A Turing language data structure implementing a linked list as an array of records

The expressions used in the binding contain references to ***pos***, the parameter to the new bound function. These references are introduced by a '!!' symbol that distinguishes between the expression source text and references to the GVL parameter. When the ***LinkedListTextLine*** function is applied, the value of the ***pos*** parameter is textually substituted into the expression.

Each expression also has a type specifier in the application language that indicates the type of the expression. These type specifiers are introduced by a '::' symbol. Type specifiers are required when the application programming language is statically typed (such as Turing, Pascal, C, etc.) and are optional when the language is dynamically typed. Existence of these type specifiers allows the polymorphism in GVL functions to be resolved statically.

The second part of the binding is a *specialization* clause. This caluse says that in the bound version of the function, all applications of ***LinkedList*** should in fact be applications of ***LinkedListTextLine***. Function references can only be specialized to bound versions of the same function.

The following application displays the array data structure as a linked list

$$\delta \text{ "currentLine :: lineReference" : LinkedListTextLine}$$

The binding process can be modeled as a second order function over GVL functions. We define the second order function ***bind*** such that

$$bind : GVL\ function \times parmList \times idList \times specializationList \rightarrow GVL\ function$$

where the first ***GVL function*** is the function to be specialized, the ***parmList*** is a list of values to which the parameters of the function are to be bound, the ***idList*** is the list of formal parameters to the new function, and the ***specializationList*** indicates what functions are to be replaced in the definition of the function being bound. The definition of ***bind*** can then be expressed as follows:

Let f' be defined as

$$bind\ (f, (b_1, \ldots, b_j), (i_1, \ldots, i_k), ((f_1, f_1')), \ldots, (f_n, f_n')))$$

then

$$\begin{aligned} f'(a_1, \ldots, a_k) = \\ \quad & f[f_1/f_1'; \ldots; f_n/f_n'] \\ & \quad (b_1[!!i_1/a_1; \ldots; !!\, i_k/a_k], \\ & \quad \ldots \\ & \quad b_j[!!\, i_1/a_1; \ldots; !!\, i_k/a_k]) \end{aligned}$$

That is, the bound function is defined as the original function where all applications of specialized functions are replaced, and where each formal in the bound function

(the i_i) is replaced by the corresponding actual parameter to the bound function (the a_i) in the actuals to the bound function (the b_i).

For example, the binding and application of the ***LinkedList*** display function can be expressed as follows

```
LinkedListTextLine (pos) is defined as
  bind (LinkedList,                                    (the function being bound)
        ("lines (!!pos).currentLine::string",
         "lines (!!pos).nextLine::lineReference",
         "!!pos = terminator::boolean"),               (the actuals to the bound function (b_i))
        ("pos"),                                       (the formal of the bound function (i_i))
        (LinkedList/LinkedListTextLine)                (the specialization list-replace all
                                                         uses of LinkedList with
                                                         LinkedListTextLine)
  )
```

So that the application

```
LinkedListTextLine ("currentLine")
```

has the meaning

```
LinkedList [LInkedList/LinkedListTextLine]
  ("lines (!!pos).currentLine::string"          ["!!Pos"/"currentLine"],
   "lines (!!pos).nextLine::LineReference"      ["!!pos"/"currentLine"],
   "!!pos = terminator::boolean"                ["!!pos"/"currentLine"])
```

by substituting the actual parameter "***currentLine***" into the definition of ***LinkedListTextLine***.

14. Other Examples

This section presents three more examples of output specifications to illustrate additional points about GVL.

14.1. Doubly-linked List

Figure 11 shows the display of a doubly-linked list of strings as the list is constructed. Figure 12 shows the GVL function used to display this list. The ***DoubleLinkedList*** function is similar to that used to display a singly linked list: a ***cond*** function tests for the end of the list; at the end, a terminator box is drawn. The list is drawn by drawing the current element using the parameter function ***DisplayElementType***. An arrow is drawn to the list on the left, and another to the list on the right. Both of these are doubly-linked lists also, and are therefore displayed recursively.

Because of the recursive description of the left and right references as doubly-linked lists, the specification is infinitely recursive. When a given element (e.g. *'Jeremy'*) is drawn, the element to its right (*'Eileen'*) is drawn as a doubly-linked list. The *'Elieen'* element redraws the element to its left (*'Jeremy'*) as a doubly-linked list, thereby starting an infinite recursion. As was seen in the singly-linked list example, the recursion is resolved by drawing repeated instances of the display in the same location as the original.

This detection and resolution of infinite recursion is a powerful specification device. The ***DoubleLinkedList*** function is actually a specification of a general binary graph, where the display shows the current node, and edges connecting to up to two other

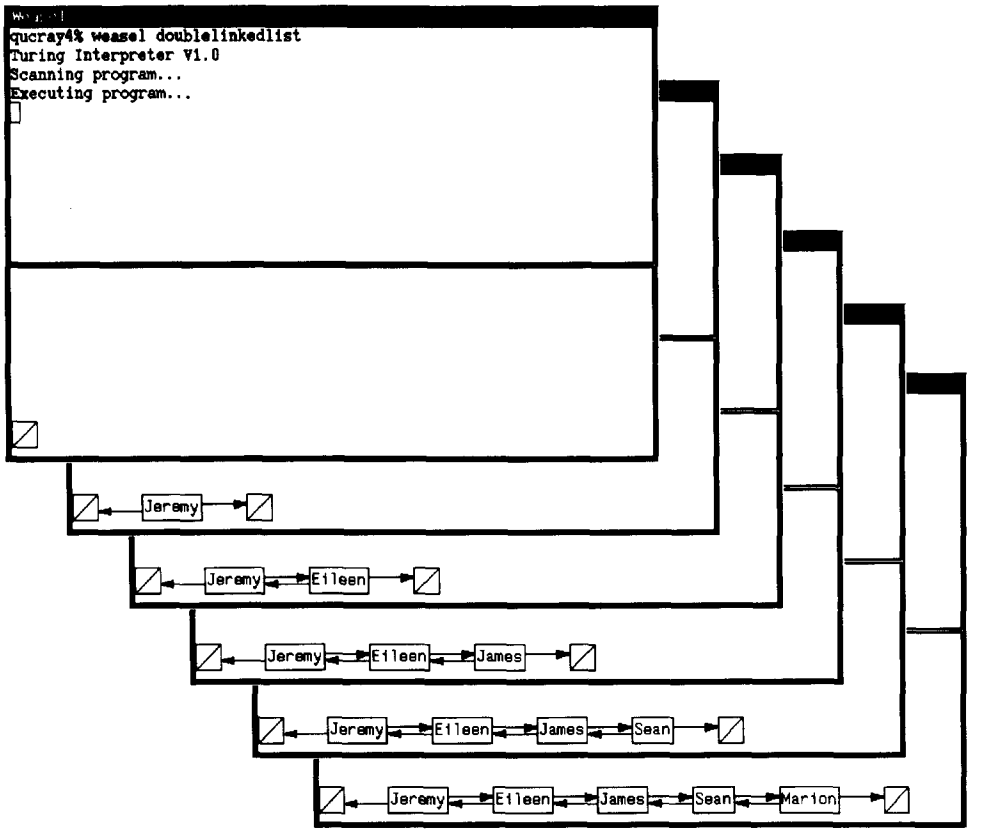

Figure 11. A series of snapshots over time of a doubly-linked list being constructed

nodes in the graph. This specification can be extended to handle n-ary graphs by adding additional links.

14.2. Bar Graph

The second example is an output specification to draw a bar graph. Figure 2 shows the output when a bar graph specification is applied to a list of real numbers. Figure 13 shows the GVL functions that produced this output. To show how specifications can be built using a number of layered display functions, three functions are used to construct this example. At the highest level, a bar graph is a list of labeled bars. Each labeled bar is in turn a bar with a string label underneath it.

Applications within abstract functions pass only one parameter, which may be

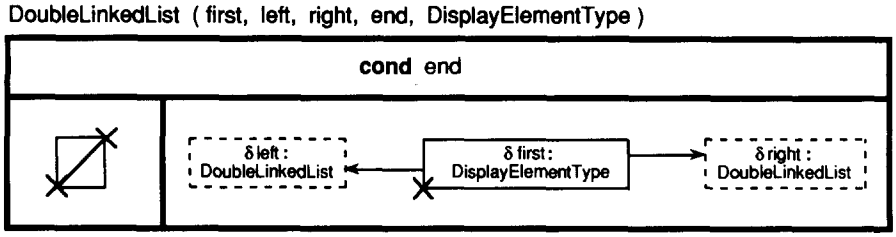

Figure 12. GVL function to display a doubly-linked list

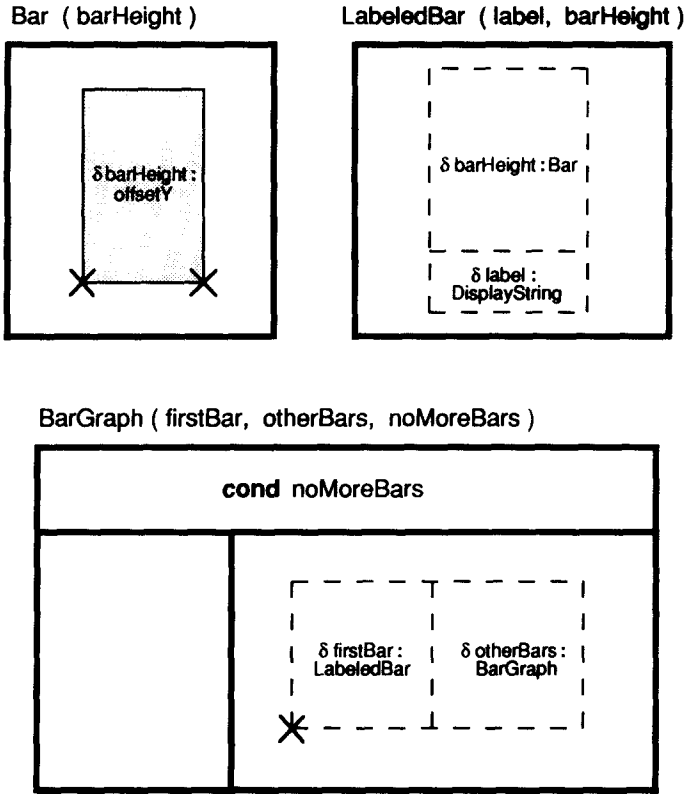

Figure 13. GVL functions to draw the bar graph shown in Figure 2

matched to several in the function being applied. These inconsistencies must be resolved when the abstract functions are bound. For example, in the **BarGraph** function, the first bar is displayed as a ***Labeledbar***, while ***LabeledBar*** requires two parameters. Before ***BarGraph*** can be applied, either the ***firstBar*** parameter must be bound to a list of two expressions, or the ***LabeledBar*** function must be specialized to a function taking only one parameter. (To produce the given output, the first option was used.)

The ***Bar*** function draws a filled box of fixed width. The box surrounds an application of the ***offsetY*** function. This function draws an invisible line of the height specified by its parameter. The filled box is sized to surround the parameter, and is therefore made the required height. The ***offsetY*** function (and the corresponding ***offsetX*** function) can be defined recursively in terms of the primitives already given. In the current implementation these functions are built in, and are implemented in a more efficient manner.

14.3. Game Boards

Our final example illustrates the strong potential for generic re-use inherent in the functional specification style. Figure 14 gives a set of GVL functions for drawing a grid-like gameboard which might be the output of a Tic-Tac-Toe playing program.

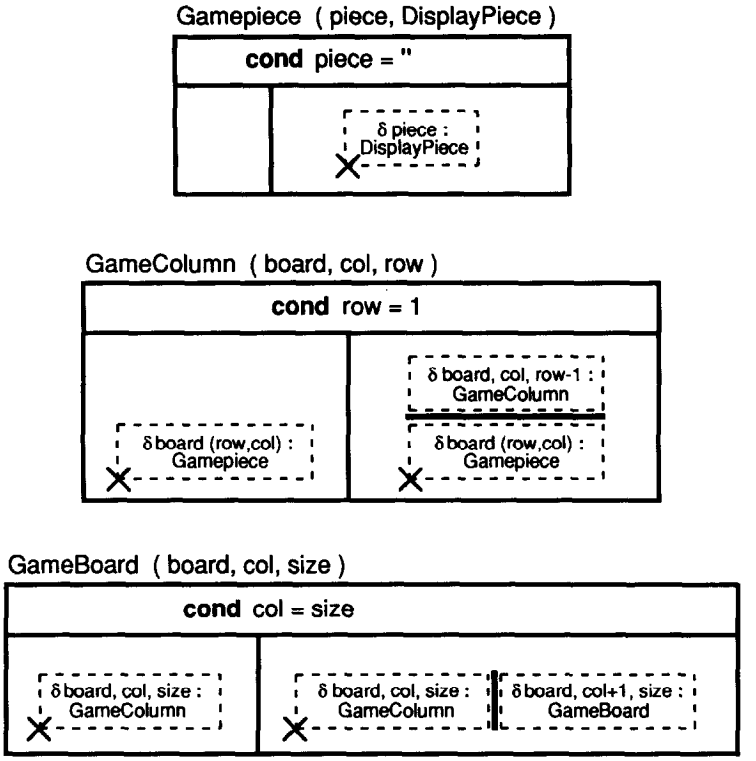

Figure 14. GVL functions to draw a generic grid-like game board

The specification consists of a number of mutually recursive layered functions. The gameboard is specified as a horizontal sequence of gameboard columns separated by bold vertical lines, where each gameboard column is specified as a vertical sequence of game pieces separated by bold horizontal lines. The function ***DisplayPiece*** has intentionally been left unspecified so that we can bind it to GVL functions for various kinds of game pieces, depending on the game the program plays.

Unlike our previous examples, this set of functions is already partially bound to a particular kind of application program data structure. In this case the functions assume that the application program will be using an $N \times N$ matrix to represent the game board, which will be bound to the parameter 'board' when the specification is used. This specification may seem awkward when compared to the direct manipulation style of tools such as HyperCard [9]. As we shall see, however, the GVL functional specification is flexible and re-usable in ways that the simple direct drawing of the Tic-Tac-Toe board cannot hope to be.

Figure 15 shows the result of binding the gameboard specification to a simple 3×3 Tic-Tac-Toe playing program, with the function ***DisplayPiece*** bound to the new GVL function ***XOpiece*** shown in the figure. Because the GVL gameboard specification is a functional visual specification of how to draw gameboards in general rather than simply a drawing of a 3×3 one, this same binding of the specification automatically adapts to changes in the board size: if the Tic-Tac-Toe program changes

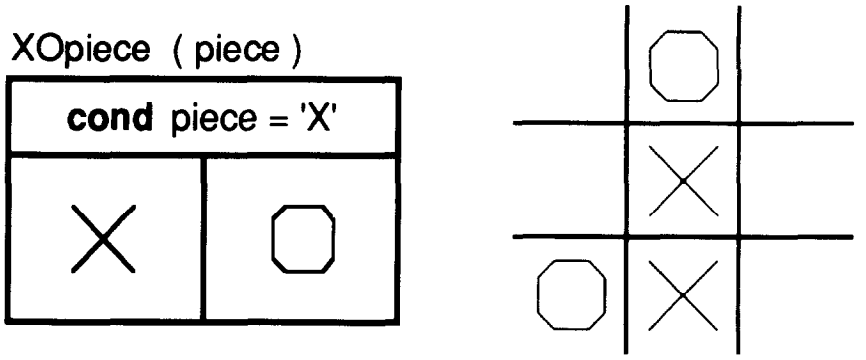

Figure 15. A GVL function to draw the pieces of a Tic-Tac-Toe game, and the result of binding the **Gameboard** function of Figure 14 to a 3 × 3 Tic-Tac-Toe playing program

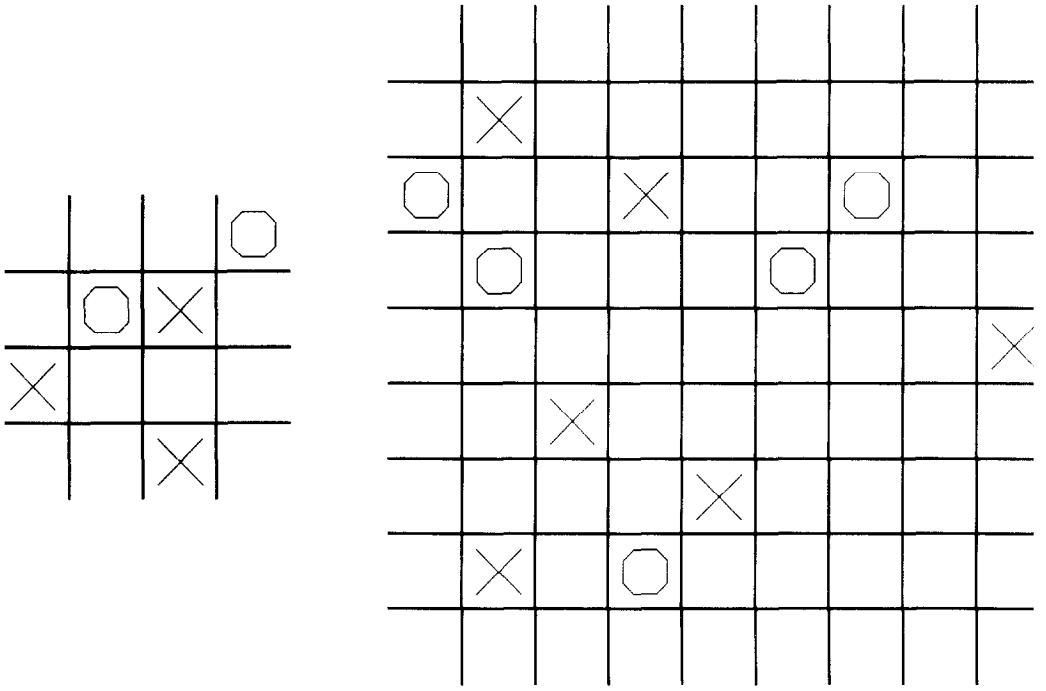

Figure 16. The result of changing the size of the board in the Tic-Tac-Toe playing program

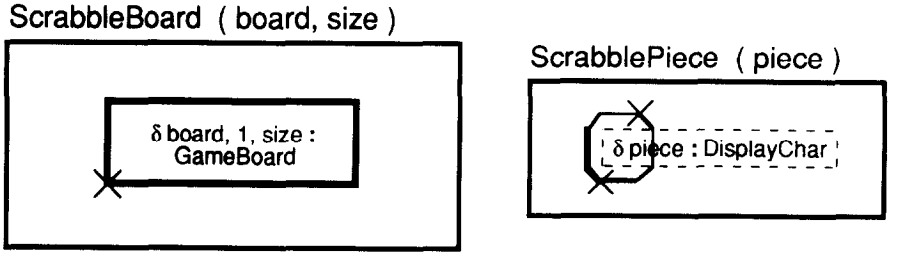

Figure 17. A GVL function to draw the pieces of a Scrabble game, and the use of the **GameBoard** function to specify a new function for drawing a Scrabble board

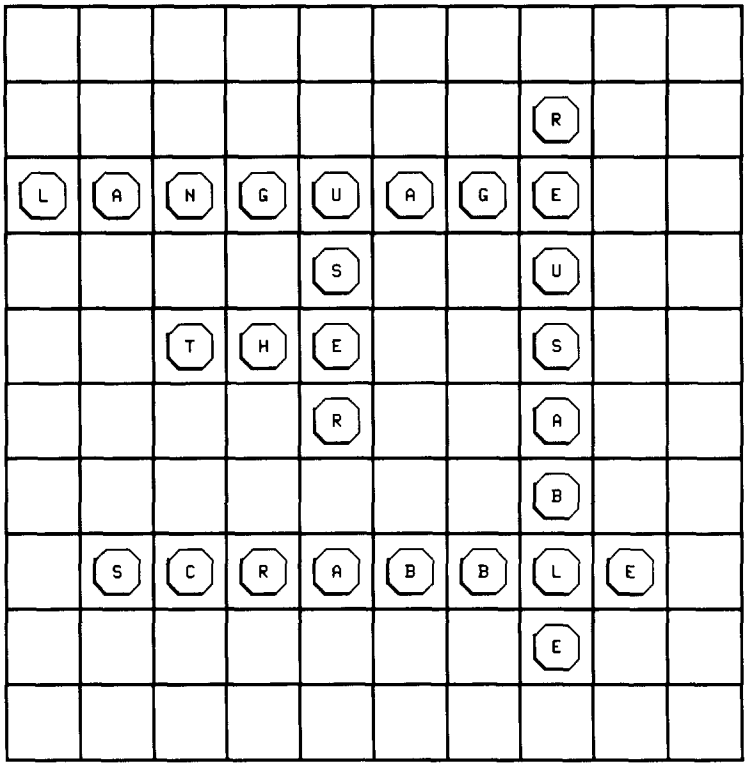

Figure 18. The result of binding the **ScrabbleBoard** function of Figure 17 to a Scrabble playing program

to use a 4×4 or 9×9 board, the *unchanged* GVL specification automatically adapts, as shown in Figure 16.

Adapting the gameboard specification of Figure 14 to drawing a Scrabble board is similarly easy. Since Scrabble pieces consist of arbitrary letters enclosed in a tile, we bind the ***DisplayPiece*** function to a new function, ***ScrabblePiece***, which specifies that (Figure 17). And since Scrabble boards normally have a border to the grid, we invoke the ***GameBoard*** function from within a new GVL function ***ScrabbleBoard*** that draws a bold border around the gameboard (also in Figure 17). The result of binding this combined specification to a Scrabble-playing program is shown in Figure 18. As before, the specification automatically adapts to changes in the board size.

15. Implementation

The *Weasel* environment is a prototype implementation of the conceptual view model. The environment consists of a front end that deals with the user, and a back end that

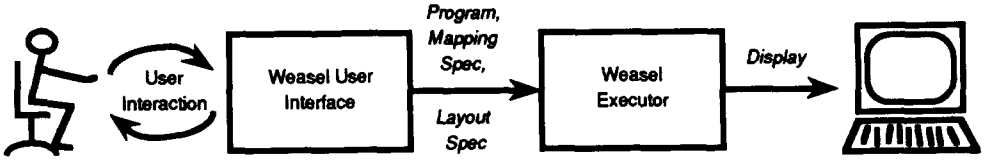

Figure 19. High level structure of the Weasel prototype implementation of the conceptual view model

executes the application program and displays views (Figure 19). The back end has been implemented, and was used to generate all of the output shown in this paper. (The implementation of the back end is described in detail in an earlier report [4]. A protoype front end has been designed [10] but is as yet unimplemented.)

The front end (the Weasel User Interface) consists of a number of loosely integrated tools. A graphical editor allows the user to draw and edit conceptual view specifications. A binding tool allows the specifications to be specialized to a particular data structure. Specifications can be placed in a library for later re-use. A text editor is used to allow the programmer to develop the application program.

The back end (the Weasel Executor) interacts with the execution of the application program, and applies the bound form of conceptual view specifications to the running program. Currently, two versions of the Executor have been implemented; the first interprets application programs written in the Turing language [5], the second is based on compiled applications written in Turing Plus.

The interface between the front and back end consists of three components: the application program to be executed, a set of conceptual view specifications written in GVL, and control information to guide the binding and execution. Conceptual view specifications are presented in a textual form generated by the front end from the graphical representation. The translation is straightforward: for each primitive in the graphical mapping language, there is a corresponding textual primitive. The semantics of this textual language and the details of the translation process are described in an earlier report [4]. Since the implementation of the Weasel Interface is incomplete, the examples of output shown in this paper were generated by entering the textual version of the GVL functions directly.

16. Related Work

The development of the conceptual view model and the GVL specification language was influenced by a number of different sources. Shaw has proposed a new model for input and output in programming languages [11]. This model recognizes the need for an output model to support two-dimensional display devices and the output of abstract data types. The conceptual view model extends some of these ideas by demonstrating how such a model might be realized.

Systems to aid in designing and implementing user interfaces are generally referred to as User Interface Management Systems (UIMSs); a good overview of the field can be found in Pfaff [12]. The conceptual view model can be thought of as describing the output component of a UIMS.

Other language systems have used methodologies similar to the conceptual view model. Both the Smalltalk Model View Controller (MVC) methodology [13] and the use of active values in Loops [14] allow the user to encode conceptual views in the application programming language.

Work in tools to aid in program debugging [15, 16], program visualization [17, 18] and systems to support graphical output [1, 19] have contributed toward the design of the conceptual view model. The two stage functional model used by Roman and Cox [20] in their work on visualizing concurrent computations is similar to the conceptual view model's three stages, although their 'intervention semantics' lacks an independent synchronization criterion.

There have been a number of earlier graphical languages used in different problem domains. The ThingLab system included a graphical sublanguage used to express graphical constraints [21]. The Pict programming language is an imperative language based on flowcharts [6]. FPL is a graphical representation of the Pascal language [22]. Other such languages are described by Myers [2].

The advantages of applying a graphical notation to functional languages have been demonstrated by a number of other systems. The Prograph language and programming environment [23] is based largely on FP and provides an elegant variable-free syntax. In this syntax, data flow is represented using directed arcs, but the functional nature of the language means that these arcs are restricted and therefore more readable than Prograph's imperative counterparts. The Show and Tell system [24] is also based on a declarative, hierarchical box syntax. The language is relational, where parts of the program are left unspecified and are filled in as a result of 'execution'. Cardelli [25] has demonstrated a graphical syntax for a language similar to ML [26].

Other languages have been tailored to expressing output mappings. The FDL language used in the VIPS debugger [15] is a textual, Ada-like language with support for output. The GRINS language [27] is a textual language used to express program input and output. Both of these languages, while introducing the flexibility of a language tailored to output, have the disadvantage of trying to encode two-dimensional information into a one-dimensional notation.

Other systems have used graphical languages designed to express output. The Descartes system [1] allows the user to draw the desired output. It does not, however, contain any mechanisms for selection or repetition. The Garden environment [19] allows the user to draw output using a set of high-level primitives. There is no facility to build new primitives, however. The authors, for example, point out that it would be impossible to express the display of an array data structure in Garden.

The Peridot user interface management system [2] also uses a graphical language to express output. These facilities include *iterations* and *conditionals* to support repetition and selection. Because it is intended to be a convenient interface in a UIMS, Peridot's language is intentionally higher level and hence less flexible than the language presented in this paper. For example, iterations are tied ultimately to Lisp lists, and operate at one level only. There is no way of naming a part of a specification and using it recursively.

17. Conclusion

This paper has described a visual functional language designed to support the conceptual view model of output. This model of output is based on the separation of the output specification of a program from the program itself, and the use of implicit synchronization to allow the data state of the program to be continuously mapped to a display view. The visual functional language GVL is used to express this mapping from the program's data state to the display.

It was argued that a notation with the full power of a programming language is necessary for this task, and it was shown that a functional language using graphical notation allows compact, convenient specification of data structure mappings. High level features such as infinite recursion detection were shown to simplify these specifications.

Earlier graphical programming languages have had problems that flow of control can become tangled in a graphical system, that variables are hard to represent graphically, and that it can be hard to combine two graphical programs with certainty that they do not interfere with each other. These problems are avoided using the functional paradigm.

18. Acknowledgements

The back end of the Weasel prototype was developed by the authors as part of the Programming Language Technology project at Queen's University, Kingston, Canada. Troy Spetz has designed a front end for the Weasel environment [10]. Stefan Hügel has implemented the Weasel binding tool. The development of the protoype was greatly aided by the helpful support of Mark Mendell and the Turing language group at the University of Toronto. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada and the Information Technology Research Centre.

Work on Weasel, GVL and the conceptual view model is still in progress. At GMD Karlsruhe, GVL is bieng extended to support full user interface construction with the addition of interactor-based input functions, user-specifiable layout styles and automatic synchronization of mixed-control user interfaces without explicit programming by the user. At Queen's University, the graphical output capabilities of Weasel/GVL are being extended to support automatic program visualization, and in particular automatic non-intrusive animation of concurrent systems, and GVL capabilities are being integrated into a dialect of the Turing programming language as built-in language features. At IIEF Berlin, GVL is being adapted to provide view programmability to the CAS system for generating documentation-quality views of VLSI circuits [28].

References

1. M. Shaw, E. Borison, M. Horowtiz, T. Lane, D. Nichols & R. Pausch (1983) Descartes: a programming-language approach to interactive display interfaces. *SIGPLAN Notices* **18**(6), 100–111.
2. B. A. Myers (1987) Creating user interfaces by demonstration. Technical report CSRI-196. Computer Systems Research Institute, University of Toronto, Canada.
3. T. C. Nicholas Graham & J. R. Cordy (1989) Conceptual views of data structures as a model of output in programming languages. In: *Proceedings of HICSS-22, Hawaii International Conference on Systems Sciences.* Hawaii, pp. 1064–1074.
4. T. C. Nicholas Graham (1988) Conceptual views of data structures as a programming aid. Technical report 88–225. Department of Computing and Information Science, Queen's University at Kingston, Canada.
5. R. C. Holt & J. R. Cordy (1988) The Turing programming language. *Communications of the ACM* **31**, 1410–1423.
6. E. P. Glinert & S. L. Taminoto (1984) PICT: an interactive graphical programming environment. *IEEE Computer* **17**(11), 7–25.
7. S. L. Peyton-Jones (1986) *The Implementation of Functional Programming Languages* Prentice-Hall, London.
8. E. C. R. Hehner (1984) *The Logic of Programming* Prentice Hall International, Englewood Cliffs, New Jersey.
9. Apple Computer (1987) *The HyperCard User's Guide* Apple Computer Inc., Cupertino, California.

10. T. D. Spetz (1990) WeaselUI: a user-interface design for the weasel programming environment. M.Sc. Thesis. Department of Computing and Information Science, Queen's University at Kingston, Canada.
11. M. Shaw (1986) An input–output model for interactive systems. In: *Proceedings of CHI 86: Conference on Human Factors in Computing Systems.* Boston.
12. G. E. Pfaff (ed.) (1983) *User Interface Management Systems* Springer-Verlag, Berlin.
13. G. E. Krasner & S. T. Pope (1988) A cookbook for using the model-view-controller interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming* **1**(3), 26–49.
14. M. J. Stefik, D. G. Brobow & K. M. Kahn (1986) Integrating access-oriented programming into a multiparadigm environment. *IEEE Software* **3**(1), 10–18.
15. S. Isoda, T. Shimomura & Y. Ono (1987) VIPS: a visual debugger. *IEEE Software* **8**(3), 8–19.
16. A. Myers (1983) Incense: a system for displaying data structures. *Computer Graphics* **17**(3), 115–125.
17. M. H. Brown (1988) Perspectives on algorithm animation. In: *Proceedings of CHI 88, Conference on Human Factors in Computing Systems.* Washington, D.C., pp. 33–38.
18. M. H. Brown & R. Sedgewick (1985) Techniques for algorithm animation. *IEEE Software* **2**(1), 28–39.
19. S. P. Reiss & J. N. Pato (1987) Displaying programs and data structures. In: *Proceedings of HICSS-20, Hawaii International Conference on Systems Sciences.* Hawaii, pp. 391–401.
20. G.-C. Roman & Kenneth C. Cox (1989) A declarative approach to visualizing concurrent computations. *IEEE Computer* **22**(10), 25–36.
21. A. Borning (1986) Defining constraints graphically. In: *Proceedings of CHI 86, Conference on Human Factors in Computing Systems.* Boston, pp. 137–143.
22. N. Cunniff, R. P. Taylor & J. B. Black (1986) Does programming language affect the type of conceptual bugs in beginners' programs? A comparison of FPL and Pascal. In: *Proceedings of CHI 86, Conference on Human Factors in Computing Systems.* Boston, pp. 175–182.
23. T. Pietrzykowski & S. Matwin (1985) Prograph: a preliminary report. *Computer Languages* **10**, 91–126.
24. T. D. Kimura, J. W. Choy & J. M. Mack (1986) A visual language for keyboardless programming. Technical report 86-6. Washington University, Washington.
25. L. Cardelli (1983) Two-dimensional syntax for functional languages. In: *Proceedings of Integrated Interactive Computing Systems,* pp. 107–119.
26. R. Milner (1985) The standard ML core languages. *Polymorphism* **2**(2), 1–28.
27. D. R. Olsen Jr., E. P. Dempsey & R. Rogge (1985) Input–output linkage in a user interface management system. Proceedings of SIGGRAPH '85. *Computer Graphics* **19**(3), 225–234.
28. A. Iwainsky, S. Kaiser & M. May (1990) Computer graphics and layout design in documentation processes. *Computers and Graphics* **14**(3), 127–135.