AUI: A Programming Language for Developing Plastic Interactive Software

Kevin A. Schneider
Department of Computer Science
University of Saskatchewan
Saskatoon, SK S7N 5A9 Canada
kas@cs.usask.ca

James R. Cordy
Department of Computing and Information Science
Queen's University
Kingston, ON K7L 3N6 Canada
cordy@cs.queensu.ca

Abstract

With the proliferation of consumer computing devices with varied display and input characteristics, it has become desirable to develop interactive systems that are usable across multiple physical environments without requiring costly redesign and reimplementation. Interactive software that easily adapts to new computer systems and environments while maintaining its usability is said to be 'plastic'. This paper introduces the AUI programming language that was designed specifically to support the development of plastic interactive software.

An AUI program describes the abstract interaction of the user interface, independent of a particular physical device or concrete interaction style. The features of the AUI language are presented here with examples of how they are used to specify user interaction. As well, this paper describes a prototype implementation that uses function application, pattern matching and lazy evaluation techniques to process the abstract descriptions of the display and user actions.

1. Introduction

In 1999, David Thevenin and Joëlle Coutaz outlined a framework and research agenda that introduced the notion of user interface *plasticity* [17]. Plasticity addresses the requirement that an interactive system be accessed from a variety of physical devices including "dumb" terminals, personal computers and handheld computers. The desire is to specify the interactive system once while preserving its usability across the various physical environments, and at the same time minimizing development and maintenance costs. A plastic interactive system is one that adapts to a wide range of user interface styles. As consumer computing devices with varied display and input characteristics become more common, the need for plastic interactive software becomes more desirable.

The AUI language was designed for specifying plastic interactive systems. The concrete, platform and environment, characteristics of a user interface are factored out, leaving the abstract interaction to be specified in the AUI language. Concrete user interface styles are bound to the AUI to complete the interactive system. The intent is for the interaction semantics to be described by the AUI language, and yet allow user interface designers the freedom to attach usable, platform specific, interfaces with minimal additional effort.

The AUI language is based on a model that considers an interactive system to be composed of a concrete user interface (CUI), an abstract user interface (AUI) and a functional core (computation) (cf. Figure 1). The CUI addresses issues such as input events and display updates. The AUI describes the logical elements of a user interface, the attributes of these elements and their structural relations. In addition, an AUI program describes the interaction between input, output and computation as a set of rules, or equations, that transform

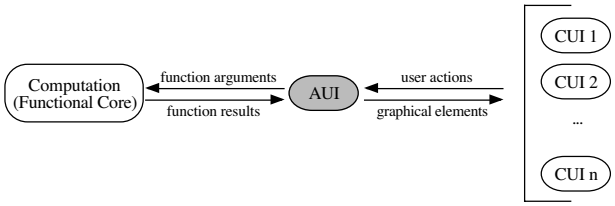

Figure 1. The AUI Model.

Multiple concrete user interfaces (CUI's) may be defined for a single abstract user interface (AUI). The combination of computation, AUI and a CUI is an interactive system.

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

the logical structure of the user interface using pattern matching and function application. The computation, or functional core, implements the application semantics. Multiple CUI's may be defined for one AUI specification. Each CUI implements a different user interface style for a particular computing platform or environment.

The binding of the abstract AUI description to concrete user interface characteristics, such as input events, display updates and toolbox directives is maintained as a separate activity, and as such is intended to address issues of user interface ergonomics. The separation of abstract and concrete user interface concerns simplifies the expression and evolution of the interaction relation and facilities a wide range of concrete user interface styles.

The goal of this paper is to describe the AUI language and a prototype implementation of an AUI compiler and runtime environment. The next section describes a simple interactive system using the AUI language to provide an overview of the approach. Section 3 describes the syntax and semantics of the AUI language in more detail, focusing on the language features that make AUI unique from similar functional languages. Section 4 describes a more complete example of using the AUI language for a simple drawing editor. Section 5 describes a version of the AUI compiler and runtime kernel that was implemented on the Apple Macintosh to demonstrate the feasibility of using the AUI approach to develop interactive systems. Section 6 describes related work, and Section 7 concludes the paper with a description of future work. An appendix of the AUI language syntax is also provided.

2. AUI Example

An AUI program is defined as a set of functions, or rules, that are applied to a logical description of the user interface. The logical description of a user interface is comprised of a display description and a sequence of user actions. The results of applying the AUI rule set is a sequence of display descriptions to be rendered by the concrete user interface.

To illustrate this concept, a simple interactive system is built in which a shaded rectangle follows the mouse pointer within a window on the display (cf. Figure 2). This example is based on a challenge put forth in the call for participation of a workshop on languages for developing user interfaces [14]: How easy is it to create a blue rectangle that follows the mouse [10, p. 143; 15, p. 156]?

The display is composed of two graphical elements, a window and a rectangle. The top, left corner of the rectangle is located on the window at an *xy*–offset from

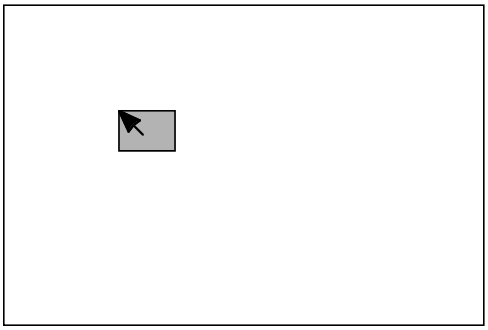

Figure 2: Rectangle Following Pointer

the top, left corner of the window. This is described in the AUI notation as follows:

window = **canvas** <300,200> {<rectangle,location>}
rectangle = **box** <35,25> (Fill Shaded)
location = <50,40>

The window is defined as a composite graphical element called a canvas. A canvas has a size and a set of <graphical element,location> tuples. The window is described as having a width of 300, a height of 200 and a single graphical element, a rectangle. The rectangle is described as a shaded box with width 35 and height 25. The location of the rectangle is 50 to the right and 40 down from the origin of the window. In this paper, pixel units are used in the examples, although in practice, more abstract units are used.

In this simple system, the only user action is the sequence of pointer positions. For example:

pointer = [<10,20>,<11,21>,<15,21>,<20,28>]

The interaction relation in which the rectangle follows the pointer is described by the function *interact*, which takes two arguments, a canvas and a pointer and returns a canvas.

interact :: CANVAS -> POINTER -> CANVAS
interact (**canvas** size {<r,l>}) p = **canvas** size {<r,p>}

The AUI notation uses pattern matching to determine if a function is evaluated. In this case, *interact* is evaluated when the canvas has a single element, *r*. The result is a canvas with the original location of *r* being replaced by the second argument, pointer location *p*.

A *main* function is introduced to apply function *interact* to the entire sequence of user actions. The function *map* is a standard function that applies a function

Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS-35'02)
0-7695-1435-9/02 $17.00 © 2002 IEEE

to every element in a sequence and returns a sequence of results.

```
main :: [CANVAS]
main = map (interact window) pointer
```

The entire AUI specification for this simple interactive system is

```
main :: [CANVAS]
main = map (interact window) pointer

interact :: CANVAS -> <N,N> -> CANVAS
interact (canvas size {<r,l>}) p =
      canvas size {<r,p>}
window = canvas <300,200> {<rectangle,location>}
rectangle = box <35,25> (Fill Shaded)
location = <50,40>
pointer = [<10,20>,<11,21>,<15,21>,<20,28>]
```

When the AUI is evaluated, the value of *main* will be

```
[
  canvas <300,200>
      {< box <35,25> (Fill Shaded)>,<10,20>>},
  canvas <300,200>
      {< box <35,25> (Fill Shaded)>,<11,21>>},
  canvas <300,200>
      {< box <35,25> (Fill Shaded)>,<15,21>>},
  canvas <300,200>
      {< box <35,25> (Fill Shaded)>,<20,28>>},
]
```

Two items are of interest with this simple interaction. First, the user interface display does not depend on its previous state. Second, the user interface has no connection to an application; that is, the entire interactive system is defined exclusively in the AUI and CUI.

If it is desirable to constrain the movement of the rectangle by the application, the connection to the application function to do so is first defined by introducing the signature of the application function (in this case *_constrain*) and composing it with the AUI expression as follows ('N' is a type synonym for the built–in type 'Number'):

```
_constrain :: <N,N> -> <N,N>
interact (canvas size {<r,l>}) p =
      canvas size {<r,_constrain p>}
```

If the application is dependent on other values, the function *_constrain* will need to be defined and applied appropriately. For example, if *_constrain*' depends on the width and height of the window, the relevant equations are:

```
_constrain' :: <<N,N>,<N,N>> -> <N,N>
interact (canvas size {<r,l>}) p =
      canvas size {<r,l'>}
l' = _constrain' <s,p>
```

The AUI notation ensures that the dependencies of the application to the user interface are explicit. Three bindings are necessary to complete the interactive system. The function *_constrain* will need to be bound to a suitable application interface. The function *pointer* needs to be bound to the mouse position in the CUI, and the CUI will need to be able to render the *main* expression. For each platform or environment for which a CUI is to built, AUI expressions will need to be rendered, and user actions will need to be translated into AUI expressions; however, the CUI / AUI translation is application independent. In addition, the AUI language features are conducive to translating AUI expressions to and from specific CUI requirements.

The next section describes in more detail the syntax and semantics of the AUI language.

3. AUI Notation

The AUI notation is based on pure functional languages such as Haskell [11], Miranda [18] and TXL [5]. The AUI language is *non-strict.* That is, an actual parameter of an AUI equation is only evaluated when its corresponding formal parameter is required. When a parameter has been evaluated, the results are saved for subsequent uses. This operational interpretation of non-strict functions is referred to as *lazy evaluation.*

The AUI language is *referentially transparent*. An expression is referentially transparent if any subexpression and the results of evaluating it can be interchanged without changing the results of evaluating the expression. That is, the meaning of an entity remains unchanged when a part of the entity is replaced by an equal part. Functional languages that observes this principle are considered to be pure functional languages: they have no side-effects. Both Haskell and Miranda are examples of pure functional languages.

3.1. AUI Specification

An AUI specification, or program, is defined by a set of equations. Equations are used to define functions and types. Evaluation of an AUI specification begins by

Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS-35'02)
0-7695-1435-9/02 $17.00 © 2002 IEEE

evaluating the first function definition. The first function is also referred to as the *main* function. Other function equations are evaluated only when and if they are needed to evaluate the first equation. The type of the first function is the abstract type of the user interface.

3.2. AUI Tokens

The tokens in the AUI language are *comments, special characters, literals, whitespace* or *identifiers*. Whitespace is any space, tab or newline.

A *comment* is preceded by a percent (%) sign and continues until a new line. An *identifier* is any alphabetic character or underscore followed by a sequence of zero or more alphanumeric characters or underscores, trailed by zero or more single quotes. Identifiers are case sensitive. That is, FOO, foo, fOo and Foo are all distinct identifiers. A *literal* is a number, character, or string.

3.3. Built–in Types

The AUI language has four built-in primitive types: Number, Char, String and Gel. A 'Gel' is a graphical element and is defined by a set of built-in constructor functions discussed in the next section. In addition to the built–in primitive types, three composite types are built–in: *list*, *set* and *tuple.*

Predefined constructors exist to represent common user interface elements, such as, colours and font families. For example, a colour may be denoted by a predefined constructor such as *Red, Green,* or *Blue.*

Interactive elements correspond to the interaction techniques in the user interface, such as, menus and palettes. In the AUI they are modelled with the built-in interactive function **choose** discussed in Section 3.5.

3.4. Graphical Element Constructors

In the AUI language the building blocks of graphical displays are referred to as *graphical elements*, or *gels* for short. Five primitive gels are defined, *label, point, box, line* and *oval,* and one composite gel is defined, *canvas.* Gels correspond to the graphics system of the concrete user interface (CUI). If text processing were supported by the CUI, additional gels could be built-in, such as *text, vertical box, horizontal box,* and *glue*. In the CUI, gels are likely to be objects. In the AUI, gels are specified using built–in constructor functions.

A graphical element is defined as either a composite graphical element, called a *canvas*, or a primitive graphical element.

A graphical element (*gel*) is one of

a. **canvas** *attributes pins*
b. *shape attributes*

A *shape* is either **oval**, **line**, **box**, **point** or **label**. *Attributes* include size attributes such as radius, width and length; position attributes such as x-offset, y-offset and angle; and graphical attributes such as pen, fill, font, style and arrows. In addition, user interface status attributes such as inactive and selected are provided for. Certain attributes only apply to certain shapes and some attributes are mutually exclusive. An alternative method of specifying a size for a graphical element is with a <*width,height*> bounding box.

Gels are placed on a *canvas* with a *pin*. A pin specifies the gel and its location. A *pin* is one of

a. < *gel* **,** *xy-offset* >
b. *gel xy-offset*

An *xy-offset* has one of the following forms:

a. < *number* **,** *number* >
b. **X** *number* **Y** *number*

An *oval* gel is specified with the built–in function **oval** followed by one or more attributes. A size attribute must be specified. In the example below an oval is specified with a radius and a fill attribute. Radius is a number and fill is one of the elements from the fill set {Clear, Shaded, Black,…}. For example the AUI expression

oval (Radius 30) (Fill Shaded)

might be represented as

An example of specifying a *line* gel begins with the built–in function **line** followed by a length, angle and an arrow attribute. Length is a number, angle ranges from 0 to 360, and arrow is a possibly empty subset of the set {AtStart, AtEnd}. An example line expression is

line (Length 15) (Angle 30) (Arrow {AtEnd})

Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS-35'02)
0-7695-1435-9/02 $17.00 © 2002 IEEE

which specifies a line 15 units long, at a 30 degree angle with an arrow at its end.

An example of specifying a rectangle begins with built–in constructor **box** followed by a <width,height> tuple and a fill attribute. For example, a clear rectangle with a width of 30 and a height of 40 is specified as

box <30,40> (Fill Clear)

A *canvas* expression defines a two dimensional Cartesian coordinate space used to position gels relative to its origin at the top left corner. The built–in constructor **canvas,** is followed by a <width,height> tuple and a set of <gel,position> tuples.

The AUI expression

canvas <300,200> {}

defines a coordinate space with width 300 and height 200 with no gels positioned on it. The following AUI expression

```
canvas <300,200> {
  (box <35,25> (Fill Shaded),<72,66>),
  (label (Style Italic) "This is a label",<160,50>),
  (oval (Radius 30) (Fill Shaded),<145,90>),
  (line (Length 15) (Angle 30) (Arrow {}),<45,150>)
}
```

may have the representation shown in Figure 3.

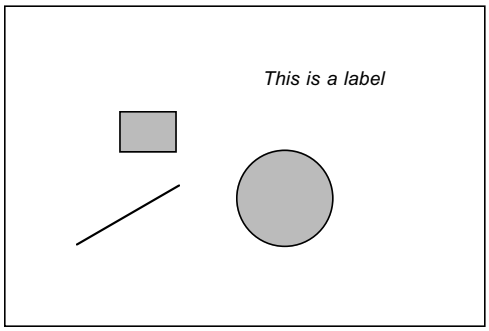

Figure 3: Example canvas Representation

3.5. Interactive Functions

Common interaction techniques such as menus, palettes, controls, buttons and text fields are modelled in the AUI by choosing from a set of possible values. For example, if a user needs to specify whether a colour should be red, green or blue the choice may be written using the built–in function **choose** as in the following AUI expression:

choose {Red,Green,Blue}

An AUI expression denotes a value. In the case of the previous expression the value may be *Red, Green,* or *Blue.* Using set comprehension, the previous **choose** may also be specified as

choose {x | x <- Colour}

The argument to **choose** may be modified with cardinality restrictions. For example, the AUI equation

arrows = **choose** 0..* {AtStart,AtEnd}

states that *arrows* will have a value with a cardinality of 0, 1, or 2. That is, the value of the expression is a member of the following set,

{{},{AtStart},{AtEnd},{AtStart,AtEnd}}

In general the syntax of **choose** is

choose [*name*] [*cardinality*] *setOrList*

Cardinality has one of the following forms

a. ***
b. *number*
c. *min* **..** *max*
d. *min* **..** ***

By default, the cardinality is '1..1'. Both *min* and *max* are numbers. A single asterisk is equivalent to '0..*' and an asterisk in '*n*..*' represents the cardinality of the *set*.

The optional *name* is an identifier and may be used to refer to the *choose* expression. In the concrete user interface a **choose** may be represented in one of many ways. For example, '**choose** {Red,Green,Blue}' may be represented as a menu, a group of check boxes or as a text field for which the colour must entered.

The *type* of a **choose** with cardinality '1..1' (the default) is the same as the elements in its **choose** set. For example, '**choose** {Red,Green,Blue}' has the type 'Colour'. When the cardinality constraint specifies a result that may have multiple elements, the type of the

Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS-35'02)
0-7695-1435-9/02 $17.00 © 2002 IEEE

choose will be the *set* of the elemental type. For example, '**choose** 0..* {Red,Green,Blue}' has the type '{Colour}'.

Since the AUI notation is non-strict, a **choose** will only be evaluated when its value is required. When a **choose** needs to be evaluated, the choice may already have been made in the CUI, and the **choose** expression will be replaced with the choosen value. When the choice has not yet been made in the CUI, the **choose** will block until the user makes a choice. Once the user makes a choice, evaluation of the AUI expression will continue.

A **choose** expression may be nested as follows:

```
ch = choose {
        choose{Red,Green,Blue}, choose{1,2,3,4}}
```

In this case, only one of the nested **choose** expressions needs to be evaluated for the value of *ch* to be determined. Possible values of *ch* include: 'Red', '1', 'Green', and '4'.

To ensure referential transparency and yet allow for different choices during an interaction for the same **choose** function, each **choose** function has an implicit numeric argument. The numeric argument is incremented at each recursive evaluation. Without this feature, the following recursive function definition (the binary operation ':' prepends an element to a list)

```
f = choose {1,2,3}:choose {1,2,3}:f
```

could only have one of the following infinite lists as a result:

(1) [1,1,1,1,…]
(2) [2,2,2,2,…]
(3) [3,3,3,3,…]

However by introducing an implicit numeric argument, *n*, the function becomes:

```
main = f 0
f n = choose n {1,2,3}:choose n {1,2,3}:f (n+1)
```

Possible values now include:

(1) [1,1,2,2,1,1,3,3,1,1,…]
(2) [3,3,1,1,1,1,2,2,3,3,…]
(3) [2,2,1,1,3,3,2,2,1,1,…]

Note that the values of the **choose** functions at each iteration will always be the same.

3.6. External Functions

External functions are used to connect to application functionality that is not specified in the functional language itself. This allows the AUI language to act as 'glue' between an application and the concrete user interface. External functions are identified by specifying their function signature, and preceding the function name by the characters "**external_**", "**ext_**" or simply "_". For example,

```
external_f :: Number -> Number
```

defines an external function that maps a numeric argument to a numeric result.

4. Drawing Editor Example

To further illustrate the AUI language, a drawing editor is defined in this section. Most AUI specifications return a **canvas** result and not a primitive graphical element, such as **oval**. In this way, multiple graphical elements may be manipulated. The following *draw* function has a **canvas** result type.

```
draw (canvas <w,h> pins) =
 choose {
  canvas <w,h> pins,
  draw (canvas <w,h> (pin:pins))
 }
 where
  pin = <g,<30,24>>
  g = choose {box <30,20> (Fill Shaded),
              oval (Radius 15) (Fill Clear),
              line (Length 10)}
 end where
```

The function *draw* takes a canvas as an argument and returns a canvas that is the same as the argument or is the same canvas with an additional *gel* positioned at <30,24>. (A *gel* with a position is referred to as a *pin*.) The *gel* may be a *box*, an *oval* or a *line*. Three possible values of applying the *draw* function to '**canvas** <184,84> {}' are:

(1) **canvas** <184,84> {}
(2) **canvas** <184,84> {
        <**line** (Length 100),<30,24>>,
        <**line** (Length 100),<30,24>>,
        <**oval** (Radius 15) (Fill Clear),<30,24>>,
        <**box** <30,20> (Fill Shaded),<30,24>>}
(3) **canvas** <184,84> {<**line** (Length 100),<30,24>>}

4.1. Choosing a Position

Instead of the graphical elements always being placed at location '<30,24>', the following revised definition of *pin* specifies that the user may choose the position of the graphical elements.

```
pin = (g,choose {<x,y>|x<-{0..w};y<-{0..h}})
```

The **choose** expression above, constructs a set of pairs of <*x*,*y*> positions that are within the size constraints of the **canvas**. If the width of the canvas is 3 and the height is 7, the following set of positions would be constructed for choosing from:

```
{<0,0>,<0,1>,<0,2>,<0,3>,<0,4>,<0,5>,<0,6>,<0,7>,
 <1,0>,<1,1>,<1,2>,<1,3>,<1,4>,<1,5>,<1,6>,<1,7>,
 <2,0>,<2,1>,<2,2>,<2,3>,<2,4>,<2,5>,<2,6>,<2,7>,
 <3,0>,<3,1>,<3,2>,<3,3>,<3,4>,<3,5>,<3,6>,<3,7>}
```

4.2. Choosing Size and Attributes

To specify that the user may also choose the size of the graphical elements, their fill, and whether or not arrows appear on the line, results in the following revised definition of *g*:

```
g = choose {
      box <width,height> (Fill fill),
      oval (Radius choose {1..50}) (Fill fill),
      line (Length choose {1..100})
                (Angle choose {0..360})
                (Arrows arrows)}
width = choose {1..100}
height = choose {1..100}
fill = choose {Clear,Shaded,Black}
arrows = choose * {AtStart,AtEnd}
```

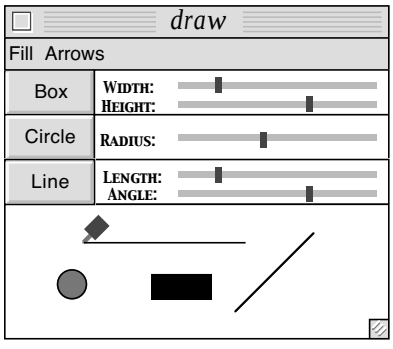

Figure 4: Example CUI for the drawing editor.

Figure 4 shows an example concrete user interface for the simple drawing editor. The size and angle choices are represented as sliders and the *fill* and *arrows* choices are represented as menus in a menu bar. A stick pin is used to position the graphical elements on the canvas.

4.3. Resizing the Canvas

To provide a means for the canvas to be resized a third choice is added to the definition of the *draw* function:

```
draw (canvas <w,h> pins) = choose {
   canvas <w,h> pins,
   draw (canvas <w,h> (pin:pins)),
   draw (canvas <choose {1..W},choose {1..H}> pins
 }
 W = 640
 H = 480
```

In the above definition, *W* and *H* refer to the width and height of the display. The resize choice may be represented in the concrete user interface as a resize control box, such as:

5. Prototype Implementation

To demonstrate the feasibility of using the AUI language to specify an interactive system, a prototype was developed and used to specify a drawing editor based on a typical commercial drawing tool. The AUI language was implemented by first translating the it into a Haskell [11] dialect called Gofer [13] and linking it to the Gofer runtime engine.

Each *choose* expression is bound to an interaction technique in the concrete user interface (CUI) and the graphical elements, *gels*, are bound to an object oriented graphics system in the CUI. The interaction techniques are routines written in C/C++ that access the Apple Macintosh toolbox and the graphics system is the Apple Macintosh imaging language, Quickdraw. The computation functions to provide the application functionality are also written in C/ C++. Input to the CUI runtime is a stream of built-in functions to be evaluated and output from the CUI runtime is a stream of *choose* values and computation function results.

The streams of choices, canvases, arguments and results are communicated between the CUI, AUI and computation through the input and output mechanisms of the implementation languages. In this way, the semantics

of each implementation language did not need to be altered. To enhance responsiveness each component has a separate thread of control and the input and output mechanisms are replaced with reads and writes to monitored queues. The queues can be thought of as channels, input/output streams or message passing facilities.

The AUI model defers the specification of temporal constraints to the interpretation of the AUI specification and its connection to the CUI. At this time temporal constraints are implicit. Three interpretations could be taken: sequential, parallel, or a combination of both. In the case of the prototype a sequential interpretation is used. That is, the order a user chooses a value, dictates the evaluation.

5.1. AUI Translation

Since most of the AUI language features are directly supported by a functional language such as Haskell, only the communication between the AUI, CUI and computation is discussed here. The communication between the AUI and the CUI is modelled as a stream of values. For example the AUI expression

fill = **choose** {None,White,25%Gray,
50%Gray,75%Gray,Black};

may have the following stream of values

[Black,White,None,75%Gray]

In general a **choose** is bound to a stream of values where the following holds.

choose min..max S =
[s | s in powerset(S) and min ≤ |s| ≤ max]

Communication to the computation functions is also modelled with streams. Each element of a stream of arguments is passed to the computation function and its result is packaged into a stream of results.

5.2. CUI Interaction Techniques

To build the CUI, each of the *choose* expressions are associated with an interaction technique. In the prototype interactive elements that are available in the Macintosh toolbox, such as menus, are used. Each of the interaction techniques are placed in an event loop that processes the mouse and keyboard events. Depending on the event, the appropriate *choose* interaction is called.

Each *choose* routine will enqueue the appropriate choice. A queue is assigned to each *choose*. The interaction technique in the concrete use interface enqueues the choice on the queue assigned it. For example, *chooseTool* will enqueue the tool choice made by the user. The AUI runtime will dequeue the choice when needed. Binding an concrete interaction technique to a *choose* expression is accomplished quite simply by using the same queue.

5.3. Graphical Element Rendering

The stream of canvas values is monitored by a *render* function in the CUI. The canvas value is translated into calls to Apple Quickdraw routines. Figure 5 is the screen image of the prototype implementation of the simple drawing editor.

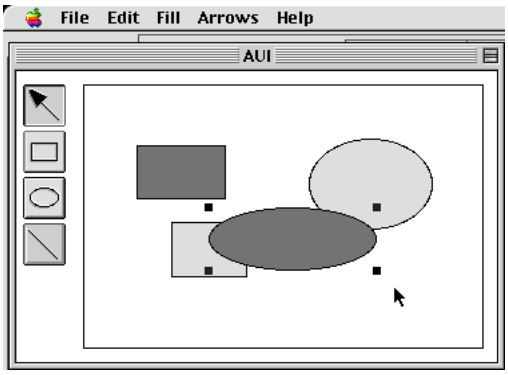

Figure 5: AUI Prototype.

5.4. Summary

This section presented a prototype compiler and runtime environment for the AUI language. The functional language equivalent of the AUI specification is interpreted by a run–time engine in a prototype implementation. Communication between CUI, AUI and computation in the prototype is through input and output facilities which have been modified to read and write to monitored queues.

6. Related Work

The AUI approach separates concrete issues of an interactive system, such as presentation, from abstract issues of an interactive system, such as domain

Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS-35'02)
0-7695-1435-9/02 $17.00 © 2002 IEEE

functionality. Similarly, markup languages, such as XML [4], WML [9] and UIML [1], separate the concrete presentation of a document from its logical description. XML is intended for the communication of structured documents, WML is intended to describe documents for wireless appliances, and UIML is intended to describe documents for a wide range of appliances including personal computers with web browsers, hand held computers and cell phones. Each style of user interface is often document and forms oriented. A domain specific language for clearly separating presentation in forms based user interfaces has also been designed [2]. The AUI approach differs in that a wider range of user interface styles is supported, including user interfaces with direct manipulation of application objects as exemplified by a graphical drawing editor.

Also related to the AUI approach are user interface generators. Automatic generation of user interfaces involves defining one or more models from which a graphical user interface is generated. Rules are specified as to how the defined model or models are to be translated into the graphical user interface. The designer may be allowed to specify one or more models, such as a domain model, a data model, a dialogue model, a platform model and a task model.

Model approaches include UIDE [3], HUMANOID [16], ADEPT [12], TRIDENT [19], Fran [6] and Teallach [8]. Most early automatic generation approaches are based on a single model and imposed a fixed set of interactors with a rigid style guide. Application specific interactors are not able to be specified.

Recent research defines additional models, however this introduces the difficulty of integrating the models. For example, the Teallach project automatically generates user interfaces for object oriented databases from three models: a domain model, a task model and a presentation model. Interactive tools are used to link the domain, task and presentation models together. Much like TRIDENT, Teallach provides both a concrete and an abstract model for the final presentation of the interactors.

To address the issue of plasticity, model approaches provide for the possibility of specifying a platform model. Unfortunately, the existing model approaches are based on predetermined software architectures that do not adapt to different platforms. The AUI approach, however, is based on a software architecture style that was designed to adapt to a wide range of concrete environments.

7. Conclusion

The AUI language provides a means of specifying abstract interaction to aid in the design and development of plastic interactive systems. The AUI language is based on a software architecture style that separates an interactive system into concrete user interface (CUI), abstract user interface (AUI) and computation. The AUI approach is conducive to producing alternative user interfaces for an application, since much of the interaction can be specified in the AUI. A textual CUI, a graphical CUI and CUIs for multiple platforms could be constructed for the same AUI.

The intent of the current research was to investigate whether the AUI approach was feasible for specifying an interactive system. A drawback, however, is that when used in conjunction with traditional programming languages, the AUI style may prove to be too much of a change in programming style. Future research, needs to address how the AUI language features can be made more accessible to developers of interactive systems.

Although the AUI research has focused on single user, non-distributed user interfaces with a direct manipulation style, the techniques are not necessarily limited to that domain and may be found useful for distributed, multi–user, multi–media or network based software. The AUI notation could be expanded to model these user interfaces.

The AUI notation can be used to specify some simple sequential dependencies, but complex temporal dependencies must be managed either by the concrete user interface or by the computation component. Other researchers have investigated notation for specifying temporal constraints using declarative languages such as *Clock* [7]. It would be interesting to supplement the AUI notation with an orthogonal notation to express temporal constraints based on this research.

The AUI language is a preliminary step in expressing abstract interaction as a foundation for building more elaborate user interfaces with rich semantic feedback and for improving the plasticity of interactive systems.

References

1. M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams and J. E. Shuster. UIML: An Appliance-Independent XML User Interface Language. WWW8, Toronto May 1999.
2. D. L. Atkins T. Ball, G. Bruns and K. Cox. Mawl: A domain–specific language for form–based services. *IEEE Transactions on Software Engineering*, June 1999, pages 334–346.
3. D. J. M. J. de Baar, J. D. Foley and K. E. Mullet. Coupling Application Design and User Interface Design Beyond Widgets: Tools for Semantically Driven UI Design. In *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems*, 1992, pages 259-266.

Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS-35'02)
0-7695-1435-9/02 $17.00 © 2002 IEEE

4. T. Bray, J. Paoli, and C. M. Sperberg-McQueen, eds. Extensible Markup Language (XML) 1.0. W3C Recommendation, 1998.
5. J. R. Cordy, C. Halpern and E. Promislow. TXL : A rapid prototyping system for programming language dialects. In *IEEE International Conference on Computer Languages*, 1988, pages 280-285.
6. C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the 1997 ACM SIGPLAN International Converence on Functional Programming*, pages 263–273, Amsterdam, The Netherlands, 9–11 June 1997.
7. T. C. N. Graham. Declarative Development of Interactive Systems. Volume 243 of Breichte der GMD. R. Oldenbourg Verlag, July 1995.
8. T. Griffiths, P. J. Barclay, J. McKirdy, N. W. Paton, P. D. Gray, J. Kennedy, R. Cooper, C. A. Goble, A. West and M. Smyth. Teallach: A Model–Based User Interface Development Environment for Object Databases. In *Proceedings of User Interfaces to Data Intensive Systems (UIDIS)*, IEEE Press, pages 86–96. 1999.
9. J. Herstad, D. Van Thanh and S. Kristoffersen. Wireless Markup Language as a Framework for Interaction with Mobile Computing and Communication Devices. In *Proceedings of the First Workshop on Human Computer Interaction with Mobile Devices*, Glasgow, Scotland, 1998.
10. R. D. Hill. Languages for the Construction of Multi–User Multi–Media Synchronous (MUMMS) Applications. In B. A. Myers, editor, *Languages for Developing User Interfaces*, 1986, pages 125–143.
11. P. Hudak, S. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain and J. Peterson. *Report on the Programming Language Haskell.* Technical Report, Yale University, USA, 1988.
12. P. Johnson, S. Wilson, P. Markopoulos and J. Pycock. ADEPT – Advanced Environment for Prototyping with Task Models. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems*, page 56, 1993.
13. M. P. Jones. The implementation of the Gofer functional programming system. Research Report DCS/RR–1030, Yale University, New Haven, Connecticut, USA, May 1994.
14. B. A. Myers, editor, *Languages for Developing User Interfaces.* Boston, Jones and Bartlett Publishers. 1992.
15. B. A. Myers. Ideas from Garnet for Future User Interface Programming Languages. In B. A. Myers, editor, *Languages for Developing User Interfaces*, 1992, pages 147–157.
16. P. Szekely, P. Luo, and R. Neches. Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design Understanding and Supporting the Design Process. In *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems*, 1992, pages 507-515.
17. D. Thevenin and J. Coutaz. Plasticity of User Interfaces: Framework and Research Agenda. In *Proceedings of INTERACT'99*. (IFIP TC.13 Conference on Human-Computer Interaction, 30th August-3rd September 1999, Edinburgh, UK), Technical Sessions, 1999, pages 110-117.
18. D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, 1985, pages 1-16.
19. J. M. Vanderdonckt and P. Berquin. Towards a Very Large Model–Based Approach for User Interface Development. In *Proceedings of the 1999 User Interfaces to Data Intensive Systems*.

Appendix. AUI Syntax

auiSpecification ::= {*eqn*}
eqn ::= *functionDef* | *functionSig* | *typeDef* | *typeSyn* | *matchEqn*
functionDef ::= *functionName* {*pattern*} = *expr*
[**where** {*eqn*} **end where**]
functionSig ::= *functionName* **::** *typeName1* { **->** *typeName2*}
typeDef ::= *typeName* { *typeVar* }**::=** *constructor* { *typeExpr* }
{ | *constructor* { *typeExpr* } }
typeSyn ::= *typeName* **==** *typeExpr*
matchEqn ::= **match** *identifier* **=** *pattern*
expr ::= *identifier* | *literal* | *functionName* { *expr* }
| *unaryOperator expr* | *expr binaryOperator expr*
| *choose* | *gel* | *expr* **, if** *expr* | *comprehension* | (*expr*)
list ::= **[** [*expr*] { **,** *expr* }] **]**
set ::= **{** [*expr*] { **,** *expr* }] **}**
tuple ::= < [*expr*] { **,** *expr* }] >
unaryOperator ::= **-** | **not** | **first** | **rest**
binaryOperator ::= *arithOps* | *logicOps* | *listOps* | *functionComp*
arithOps ::= **+** | **-** | **/** | ***** | **^** | **div** | **rem**
logicOps ::= **==** | **<=** | **>=** | **~=** | **<** | **>** | **and** | **or**
listOps ::= **:** | **++**
functionComp ::= **.**
comprehension ::= *listComprehension* | *setComprehension*
listComprehension ::= **[** *expr* | *qualifier* { **,** *qualifier*} **]**
setComprehension ::= **{** *expr* | *qualifier* { **,** *qualifier*} **}**
generator ::= *name* **<-** *listOrSetExpr*
| < *name,name* > **<-** *2-tupleListOrSetExpr*
| < *name,name,name* > **<-** *3-tupleListOrSetExpr* | …
listOrSetExpr ::= *listExpr* | *setExpr*
listExpr ::= **[** *number* **..** *number* **]** | **[** *char* **..** *char* **]** | **[** *type* **]**
setExpr ::= **{** *number* **..** *number* **}** | **{** *char* **..** *char* **}** | **{** *type* **}**
pattern ::= *identifier* | *literal* | _ | < [*pattern* { **,** *pattern* }] >
| **[** [*pattern* { **,** *pattern* }] **]** | **{** [*pattern* { **,** *pattern* }] **}**
| **(** [*pattern* { **,** *pattern* }] **)** | **(** *pattern* **)**
| *pattern* **:** *pattern* | *constructor* { *pattern* }
gel ::= **canvas** *attributes pins* | *shape attributes*
shape ::= **oval** | **line** | **box** | **point** | **label**
pins ::= **{** [pin {**,** pin}] **}**
pin ::= < *gel* **,** *xy-offset* > | *gel xy-offset*
xy-offset ::= < *number* **,** *number* > | **X** *number* **Y** *number*
attributes ::= {attribute}
attribute ::= < *number* **,** *number* > | **(Fill** *fill* **)**
| **(Arrows** *arrows* **)** | (**Font** *font*) | …
choose ::= **choose** [*name*] [*cardinality*] *setOrList*
cardinality ::= * | *number* | *min* .. *max* | *min* .. *
functionName ::= *name* | **external_***name* | **ext_***name* | **_***name*
***Note*:** *typeName*, *typeVar*, *typeExpr* and *name* are all identifiers.

0-7695-1435-9/02 $17.00 (c) 2002 IEEE