Proc. DSVIS 2001 - 6th International Conference on Document Analysis and Recognition, Glasgow, June 2001, pp. 40-59

Abstract User Interfaces: A Model and Notation to Support Plasticity in Interactive Systems

Kevin A. Schneider[1] and James R. Cordy[2]

[1] Department of Computer Science, University of Saskatchewan, 57 Campus Drive, Saskatoon, Saskatchewan S7N 5A9 Canada,
kas@cs.usask.ca

[2] Department of Computing and Information Science, Queen's University, Kingston, Ontario K7L 3N6 Canada,
cordy@cs.queensu.ca

Abstract. This paper introduces the Abstract User Interface (AUI) model and notation for specifying abstract interaction in interactive software systems with graphical, direct manipulation user interfaces. The AUI model is aimed at improving the plasticity of an interactive system. An interactive system is considered to be plastic when it is easily adaptable to concrete user interface styles. To support plasticity, an AUI specification defines the interaction between input, output and computation in terms of the abstract elements of the user interface: a relation we refer to as *abstract interaction*. Concrete characteristics of the user interface, such as events, callbacks and rendering, are deliberately factored out so that the abstract interaction relation can be exposed. Clearly defining the abstract interaction ensures that consistent interaction semantics is maintained independent of changes to the concrete user interface. To demonstrate the AUI concept, a range of user interface styles are presented for a single AUI specification of a drawing tool, and examples of commercial applications are presented.

1 Introduction

In 1999, David Thevenin and Joëlle Coutaz outlined a framework and research agenda that introduced the notion of user interface *plasticity* [16]. 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.

The abstract user interface (AUI) model and notation described in this paper, has been developed to help improve the plasticity of interactive systems. The approach is intended to be used not only in the development of new interactive systems, but also in adding plasticity to existing legacy applications. It is often the case that existing interactive systems were developed with little or no notion of plasticity.

The AUI model separates a user interface into concrete and abstract components so that a number of concrete user interface styles may be specified for a single abstract user interface. The AUI notation is an executable specification language used to define the abstract user interface. By only specifying abstract interaction once, it is hoped that development and maintenance costs will be reduced and that interaction semantics of an interactive system will remain consistent across multiple concrete user interfaces.

The primary goal of this paper is to introduce the AUI model and notation, showing how the notation is used to define abstract interaction. A secondary goal is to show how the model has been used to introduce plasticity into existing interactive systems.

The paper is organized into seven sections. Section 2 describes the AUI model and points out differences between the AUI model and other user interface models. The AUI language is presented in Section 3 using an example of a graphical drawing editor. Section 4 describes a prototype of an AUI language compiler and how it was used to implement the graphical drawing editor. Section 5 describes the application of the AUI model to existing large-scale legacy systems. Section 6 relates the AUI work to other research presented in this volume and Section 7 concludes the paper, describing future research.

2 The AUI Model

The AUI model considers an interactive system to composed of three components, the functional core of the application (or computation), the abstract user interface (AUI) and the concrete user interface (CUI). Together, the AUI and CUI form the user interface of the interactive system. To support plasticity of the user interface, multiple CUI's may be defined for one AUI specification (cf. Figure 1). By maintaining the distinction between abstract interaction and concrete interaction, it is hoped that a large portion of an interactive system can be designed, developed and evolved as a single entity and yet be usable across a wide range of platforms and concrete interaction styles.

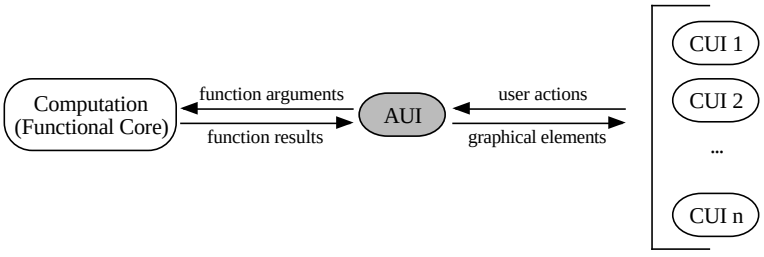


Fig. 1. AUI model. A number of concrete user interfaces (CUI's) may be defined for a single AUI. The combination of Computation, AUI and one of the CUI's forms an interactive system.

The AUI model is a synthesis of ideas from existing user interface architectural models, such as the Seeheim model [15], Arch [18], ALV [11], Interactors [14], Clock [8], TRIDENT [3] and PAC [5]. Traditionally the Seeheim model, shown in Figure 2, is used to compare user interface architectures and models.

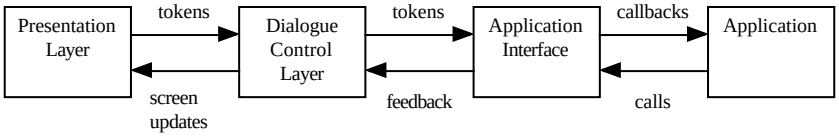


Fig. 2. The Seeheim model of user interface management systems.

The Seeheim model separates a user interface into Presentation Layer, Dialogue Control Layer and Application Interface. The Abstract User Interface (AUI) model presented here differs in that it abstracts elements from each one of these components (cf. Figure 3). Therefore, the AUI specification is not limited to describing a single Seeheim component, and can be used to describe the relation between presentation, dialogue and application interface. That is, the AUI specification defines the abstract interaction between input, output and computation.

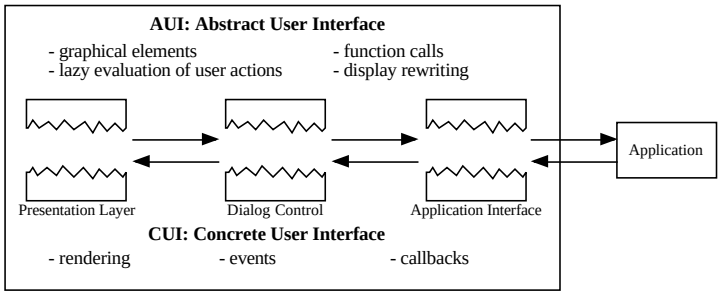


Fig. 3. Separating Seeheim components into CUI and AUI components.

The AUI model is not a replacement for the Seeheim model, but rather an orthogonal refinement of the Seeheim components into their abstract, environment independent aspects and their concrete, environment dependent ones. The refinement is explicitly aimed at improving the plasticity of interactive systems.

The AUI approach is similar to research that separates the interactor portion of a user interface into abstract and concrete components [6]. With such separation, different concrete input and output mechanisms can be used depending on user preferences, available resources or the context of the interactor. TRIDENT [3] separates presentation into abstract and concrete interaction objects

and uses matching tables to automatically transform the abstract interaction objects to concrete interaction objects depending on the specific physical target environment. The AUI research focuses on the specification of the abstract user interface. Approaches, such as that used in TRIDENT, may integrate well with the AUI model for automatically generating the concrete user interface and addressing ergonomic issues.

The AUI approach is also related to markup languages that are used to describe hypertext documents, such as XML [4], WML [10] and UIML [1]. These markup languages 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 wider range of user interface styles is supported by the AUI approach, including user interfaces with direct manipulation of application objects as exemplified by a graphical drawing editor.

2.1 Abstract User Interface Component

The abstract user interface (AUI) component, models output as a sequence of display values and input as a sequence of user actions. The AUI specification provides pattern matching rules that recursively transform the current display value and user action into a new display value (cf. Figure 4).

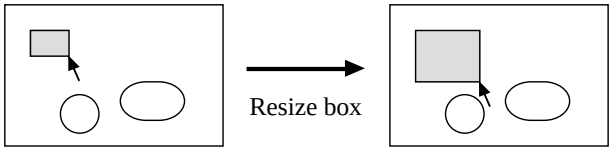
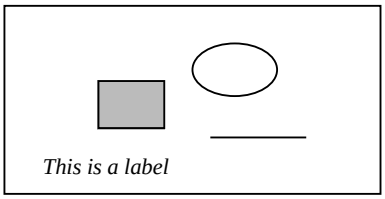


Fig. 4. Display change resulting from a user action to resize a box.

The AUI considers a display to be composed of a set of graphical and interactive elements. The graphical elements, *gels*, are specified with a set of built in constructor functions in the AUI language. Primitive *gels* have a shape, a size and an optional set of attributes for specifying characteristics, such as, pen width, fill colour and font. Some primitive *gel* shapes include *box*, *oval*, *line*, *label* and *point*.

The composite graphical element *canvas*, is used to provide position to the graphical elements. A *canvas* has a size, an optional set of attributes, and a set of <*gel,position*> tuples. A <*gel,position*> tuple is referred to as a *pin*. An example display value is shown in Figure 5.




```
canvas <200,100> {
   <box <35,25> (Fill Shaded),<50,40>>,
   <oval <45,28> (Fill Clear),<100,20>>,
   <line <50,0> (Arrows None),<110,70>>,
   <label <Font Times)(FontSize 14) (Style Italic),<20,80>>}
```

Fig. 5. Example AUI display value and its possible rendering. The display consists of a **canvas** and four pins. The **canvas**, **box**, **oval**, and **line** gels each have their size specified by a <width,height> tuple. For example, the **box** has a size of <35,25>, is shaded and is positioned on the **canvas** at <x,y> location <50,40>.

Input is abstracted as a sequence of choices. For example, selecting from a menu or palette is modelled as *choosing* from a set of values or functions. Changing a fill attribute of a rectangle could be expressed as

```
canvas <300,200> {<box <35,25> (Fill shading),<50,40>}
   where
      shading = choose {Shaded,Clear,Black}
   end where
```

Output is abstracted as a sequence of user interface expressions, each to be rendered by the concrete user interface. Evaluating the previous equations may result in one of the following values:

```
canvas <300,200> {<box <35,25> (Fill Black),<50,40>}
canvas <300,200> {<box <35,25> (Fill Clear),<50,40>}
```

Instead of having the location predefined to be <50,40>, the location of the rectangle may be chosen by the user. The function, *location*, is added to the specification to denote choosing an <*x,y*> coordinate from a set of points. Each time the user chooses a new location or shading for the rectangle a new canvas expression will be passed to the CUI for rendering.

```
canvas size {<box <35,25> (Fill shading),location}
   where
      location = choose points
      shading = choose {Shaded,Clear,Black}
   end where
```

2.2 Concrete User Interface Component

The concrete user interface (CUI) component is concerned with issues of physical layout, graphical rendering and input events. The CUI renders the AUI display values as graphical objects on the screen and produces the user action sequence. The translation of AUI input and output values to and from specific toolkits or platforms is independent of a particular interactive system. Each *choose* expression is bound to an interaction technique; however a variety of concrete representations are possible (cf. Figure 6).

Fig. 6. Possible CUI elements for the AUI expression **choose** {`Red,Green,Blue`}.

The binding of the CUI and AUI is modelled as sequences of user choices and sequences of expressions to be rendered (cf. Figure 7). For example, as the user makes menu choices over time, the choices are passed to the AUI as a sequence of menu choices. Based on the choices, a sequence of graphical expressions is passed back to the CUI. The CUI renders the graphical expressions.

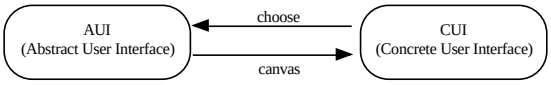


Fig. 7. Abstract user interface/Concrete user interface communication.

2.3 Computation Component

Computation is modelled as external functions in the AUI. The computation functions are used where needed in the AUI specification, and so the arguments the computation depends on are clearly expressed. The signature of the computation functions are part of the AUI specification to ensure correct typing, but the implementation of the functions is external.

Similar to the CUI binding, the binding of the AUI and computation is modelled as streams of arguments and results (cf. Figure 8). Although computation is modelled as function application, in practice the binding may update a variable or be used to bring computation in-line. The definition of the computation

may not correspond to a function in the application language but may be a procedure, a message or may correspond to the execution of a statement.

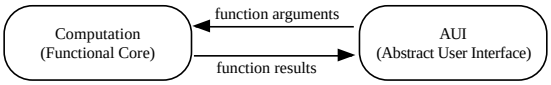


Fig. 8. Abstract user interface/Computation communication.

3 AUI Language

This section introduces the AUI notation for specifying abstract interaction using an example of a graphical drawing editor. The complete syntax of the language is provided in the appendix.

The basic framework of the simple drawing editor includes a menu bar, a tool palette, a pointer and a canvas. The AUI specification for the basic framework of the drawing editor is

```
drawingEditor = choose {tool,menuBar}
   where
      tool = choose {pointerTool,boxTool,ovalTool,lineTool}
      menuBar = choose {file,edit,fill,arrows}
      file = choose {new,open,close,save,saveAs,print,quit}
      edit = choose {undo,cut,copy,paste}
      fill = fillAttr (choose {None,White,Shaded,Black})
      arrows = arrowAttr (choose * {AtStart,AtEnd})
   end where
```

Example concrete user interfaces for the AUI specification of the drawing editor are shown in Figure 9.

The AUI language is similar to non-strict, pure functional languages like Haskell [12] and Miranda [17]. Arguments to the *choose* function are only evaluated when a value is required (lazy evaluation). When the value is required, the AUI will wait for the concrete user interface to return a value. The CUI may do one of three things:

1. A choice has already been made in the CUI and that choice is returned.
2. Although a choice has not yet been made in the CUI, the CUI returns a default choice.
3. The interaction blocks, waiting for the user to make a choice using an interaction technique. The resulting choice is returned.

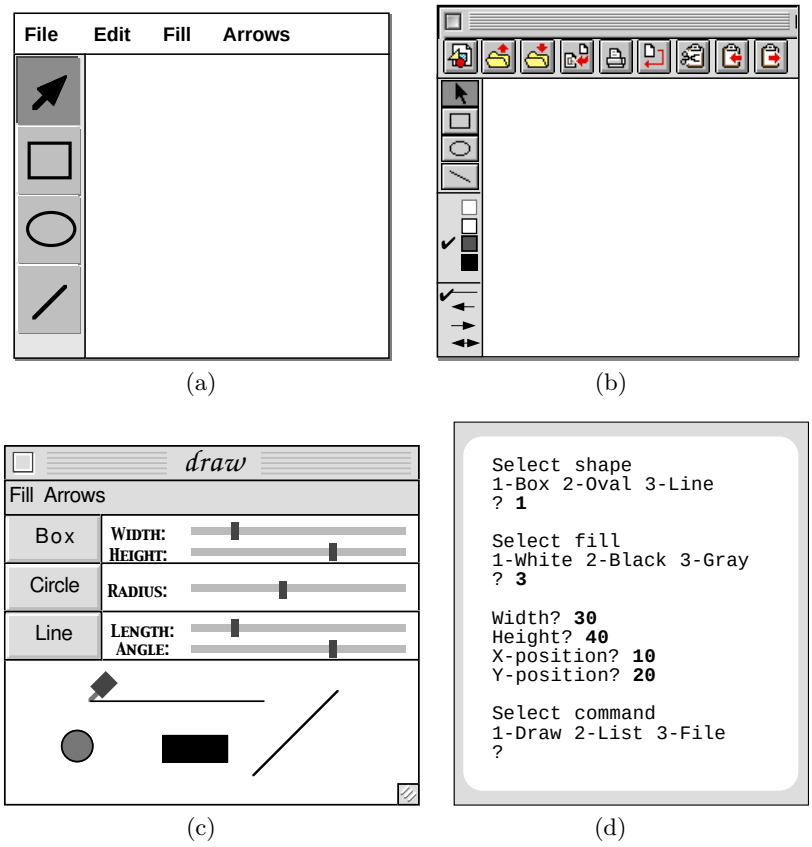


Fig. 9. Possible CUI's for the graphical drawing editor.

3.1 Modelling Canvas Interaction

To model the interaction with the drawing canvas, a function, say *draw*, is recursively applied to a canvas expression. Function *draw* is defined as a choice between leaving the canvas as is (the base case) or applying the function recursively to the *drawingEditor* function. The *drawingEditor* function will be defined in terms of the original canvas and will return a new canvas. The functions are defined as follows:

```
main = f (canvas <300,200> {})
draw c = choose {c,draw drawingEditor}
   where
      drawingEditor = ...c ...
   end where
```

3.2 Computation

Computation, or application specific functionality, is added to the AUI specification as external functions. By convention, the names of the external functions are preceded with an underscore (_). In the AUI specification only the signatures of the external functions are specified. For example, the following signatures may be defined:

```
_new :: -> Gel
_open :: String -> Gel
_save :: Gel -> Gel
_print :: Gel -> Gel
```

The external functions may then be used where appropriate in the AUI specification as long as their function signatures are respected. External functions do not necessarily correspond to procedures or functions in traditional languages but may correspond to changing a data value or transmitting a message. In the simple drawing editor a number of the file menu functions are expanded to use external functions.

3.3 Modelling the Pointer

To model drawing using a pointer requires a CUI interaction technique to accept a sequence of pointer positions, such as

```
[<100,150>,<100,155>,<100,160>,<100,170>]
```

The first point in the sequence corresponds to a point on the drawing canvas where the user first pressed the mouse button. The last point in the sequence corresponds to a point on the drawing canvas where the user released the mouse button. The intermediate points in the sequence correspond to the pointer position on the drawing canvas while the mouse button is being held and moved. In the CUI, a cursor will track the mouse movement. In the AUI, pointer interaction is defined as choosing from a set of points:

```
pts = choose * points
points = {<x,y> | x<-[0..(width c)];y<-[0..(height c)]}
   where
      width (canvas <w,h> pins) = w
      height (canvas <w,h> pins) = h
   end where
```

For example, if the canvas is defined as '**canvas** `<200,300>` {}' then the results of the function *width* will be 200 and the results of the function *height* will be 300. *Points* will have the value '{`<0,0>,<0,1>,...,<200,299>,<200,300>`}'.

Choosing from the *points* set may result in the following value which represents moving the mouse pointer from position <30,30> to position <130,130>.

```
[<30,30>,<30,31>,...,<130,129>,<130,130>]
```

3.4 Drawing

To create an object the user selects a tool from the tool bar, positions the pointer on the canvas, presses the mouse button to specify the starting point of the object and while holding the mouse button down, moves the pointer until the object is the appropriate size and then releases the button. In this way objects are specified with a list of <*x*,*y*> points.

The following functions construct a box shaped graphical element given two points from the canvas. From these two points, the size and position of the box is calculated. In a typical drawing editor, the first point, *pt1*, corresponds to where the pointer was when the mouse button was pressed and the second point, *pt2*, corresponds to where the pointer was when the mouse button was released.

```
boxPin pt1 pt2 = <gel,pos>
   where
      gel = box (gelSize pt1 pt2) attr
      pos = gelPos pt1 pt2
      gelSize <x1,y1> <x2,y2> = <abs (x1-x2),abs (y1-y2)>
      gelPos <x1,y1> <x2,y2> = <min [x1,x2],min [y1,y2]>
   end where
```

Figure 10 shows an example of pressing and holding the mouse button at position <30,30> and releasing the mouse button at <130,130>. The result box will have a width of 100 and a height of 100.

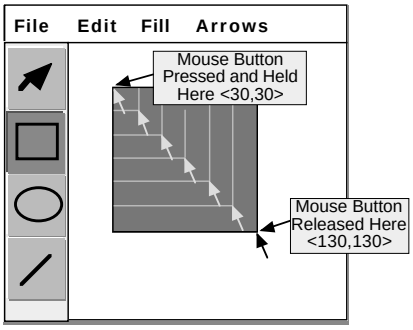


Fig. 10. Drawing a box.

Normally, graphical elements are placed on the composite graphical element *canvas*. Adding a *pin* to a *canvas* is accomplished with function, *place*.

```
place pin (canvas size pins) = canvas size (pin:pins)
```

The operator ':' inserts its left operand into the set that is its right operand. To provide rubber band feedback to the user, the following definitions are applied to the *place* function:

```
boxTool = foldc (rubberBox (first pts)) c pts
rubberBox pt1 c pt2 = place (boxPin pt1 pt2) c
```

The function *foldc* reduces a list in the following way

```
foldc f c [x1,x2,...,xn] = (...((c $f$ x1) $f$ x2)...) $f$ xn
```

Applying *foldc* to the canvas '**canvas** <192,192> {}' and the pointer sequence

```
[<30,30>,<31,31>,<31,32>,<31,31>,...,<130,129>,<130,130>]
```

results in the sequence of canvases:

```
(1) canvas <192,192> {<box <0,0> attr,<30,30>>}
(2) canvas <192,192> {<box <1,1> attr,<30,30>>}
(3) canvas <192,192> {<box <1,2> attr,<30,30>>}
(4) canvas <192,192> {<box <1,1> attr,<30,30>>}
...
(n-1) canvas <192,192> {<box <100,99> attr,<30,30>>}
(n) canvas <192,192> {<box <100,100> attr,<30,30>>}
```

Rendering the sequence of canvases will provide the rubber band feedback effect (cf. Figure 10).

3.5 Selecting

In typical graphical drawing editors, a pointer tool is provided to select one or more graphical elements for the purpose of repositioning, deleting or changing their attributes. The user selects an element by clicking on it. When a previously selected element is clicked on, the element becomes *deselected*. By holding the mouse button down while moving the pointer, the selected graphical elements are repositioned. The *pointerTool* is defined as follows:

```
pointerTool = foldc (repositionGels (first pts))
                        (selectGels (first pts) c) pts
```

The function *selectGels* selects (or deselects) the graphical element(s) that the pointer is within. The function *repositionGels* computes a new position for each of the selected graphical elements, relative to the position of the pointer. The functions *selectGels* and *repositionGels* are defined as follows:

```
selectGels xy (canvas size pins) = c'
   where
      c' = canvas size (map (selectWithin xy) pins)
      selectWithin xy pin = select pin, if within xy pin
      selectWithin c pin = pin  % otherwise
   end where

repositionGels xy1 (canvas size pins) xy2 = c'
   where
      c' = canvas size (map (reposition xy1 xy2) pins)
      reposition xy1 xy2 <gel Sel,pos> = <gel Sel,pos'>
      reposition xy1 xy2 pin = pin
      pos' = newpos xy1 xy2 pos
      newpos <x1,y1> <x2,y2> <x,y> = <x+x2-x1,y+y2-y1>
   end where

select :: PIN -> PIN  % select - turns on or off the selected gel's attribute
select <gel Sel,pos> = <gel,pos>  % deselect
select <gel,pos> = <gel Sel,pos>  % select

within :: XY-POSITION -> PIN -> BOOLEAN
within <x1,y1> <shape <w,h> attrs,<x2,y2>> = True,
        if x1 >= x2 and x1 <= (x2+w) and y1 >= y2 and y1 <= (y2+h)
within xy pin = False
```

4 Implementation

The graphical drawing editor specified in the previous section has been implemented on the Apple Macintosh platform. A concrete user interface was developed similar to the drawing component of the commercial software package AppleWorks [2]. An AUI language interpreter was developed based on the Gofer runtime engine [13]. Gofer is functional language environment for a dialect of the functional programming language Haskell [12].

The *choose* expressions are bound to common Apple Macintosh interaction technique routines written in C/C++ that access the Apple Macintosh toolbox. The graphical elements are rendered using the Apple Macintosh imaging language, Quickdraw. Computation functions are also written in C/C++.

4.1 Runtime Architecture

The communication between the CUI and the AUI, and the AUI and the computation is through the input and output streams of the functional language and the input and output features of C/C++. Input to the CUI runtime is a stream of built-in functions to be evaluated (rendered) and the output from the CUI runtime is a stream of *choose* values and computation function results.

The input/output mechanism of the AUI runtime was modified to use monitored queues, and these queues were also accessed by the CUI runtime instead of using input and output routines (cf. Figure 11). The queues can be thought of as channels, input/output streams or message passing facilities.

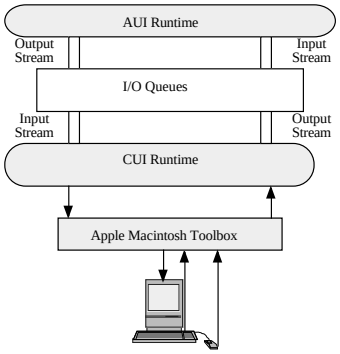


Fig. 11. Runtime Architecture.

4.2 AUI Evaluation

Since the AUI notation is based on pure functional languages, it is relatively easy to translate most of the language features directly into previously implemented functional language, such as Haskell and Miranda. As with those languages, the AUI language has structured types, pattern matching, function application, and higher order functions. And like both Haskell and Miranda, the AUI notation has a non-strict semantics (lazy evaluation) and is strongly typed.

The communication between the AUI and the CUI is modelled as a stream of values. 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 does not need to be altered. To enhance responsiveness separate threads of control are used for the user interaction and computation, and the input and output mechanisms are replaced with reads and writes to monitored queues (cf. Figure 11).

A functional language runtime engine based on Gofer [13] is provided to interpret the AUI specification. The runtime engine has been modified to communicate input and output through the monitored queues instead of through the file system.

Connecting the CUI, AUI and computation involves simply connecting the streams of choices and canvases to *enqueue* and *dequeue* routines. The interaction techniques, rendering routines and computation functions access the queues as necessary.

4.3 CUI Interaction Techniques

To build the CUI, each of the *choose* expressions are associated with an interaction technique. In the prototype interactive elements commonly available in the Macintosh toolbox, such as menus, are used. On the Apple Macintosh, each file has a *data fork* and a *resource fork*. The resource fork may contain icon, window and menu definitions. An Apple supplied visual specification tool, ResEdit, is used to draw the icons and define the menus for the CUI. Figure 12 shows ResEdit windows for defining the icons in the application. Figure 13 shows images of the menu bar and pull down menus, also defined with ResEdit.

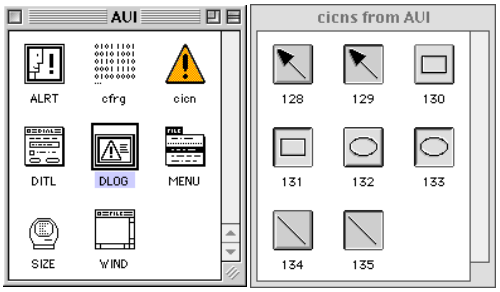


Fig. 12. The Macintosh ResEdit tool was used to define the CUI's visual resources.

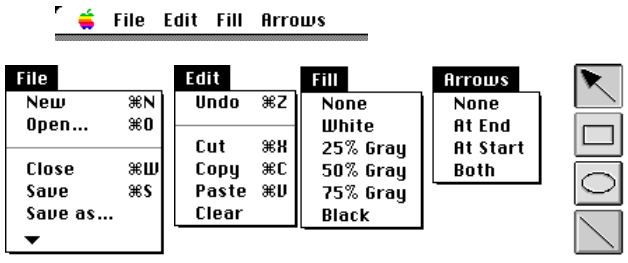


Fig. 13. Concrete interaction techniques of drawing editor.

The interaction technique in the concrete use interface enqueues the user's choice on the queue assigned it. The C/C++ code to handle a mouse event and place the point selection on a queue is:

```
GetMouse(&mouseLoc); /* Apple Toolbox Routine */
enqueue(ch9,mouseLoc.h); /* ch9 is the choose queue for pts */
enqueue(ch9,mouseLoc.v);
```

4.4 CUI 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. For example, the following C/C++ code renders a *pin* of type *line*:

```
/* c1 is the queue assigned to the canvas */
if (dequeue(c1) == linePin) {
   pin.shape = linePin;
   pin.w = dequeue(c1);
   pin.h = dequeue(c1);
   pin.x = dequeue(c1);
   pin.y = dequeue(q);
   pin.selected = dequeue(q);
   /* Apple Quickdraw routines to draw Line */
   PenMode(patCopy);
   RGBForeColor(&black);
   MoveTo(pin.x,pin.y);
   Line(pin.w,pin.h);
   if pin.selected {.../* draw handles */...}
}
```

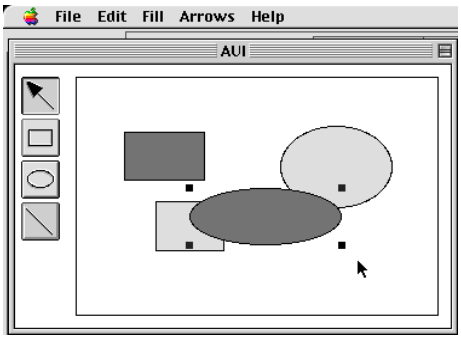


Fig. 14. Drawing editor prototype running on the Apple Macintosh.

5 Application to Legacy Systems

The AUI model has recently been used in a commercial setting for increasing the plasticity of legacy applications. In one example, the AUI model was used to separate business and user interface logic so that the business rules could be adapted to a wide range of user interfaces through a multi-channel messaging architecture (cf. Figure 15). In a second example, the AUI model was used to migrate a legacy application from a single session to a multiple session web-based architecture (cf. Figure 16). In both cases, the AUI modelled the interface between application functionality (computation) and concrete user interface.

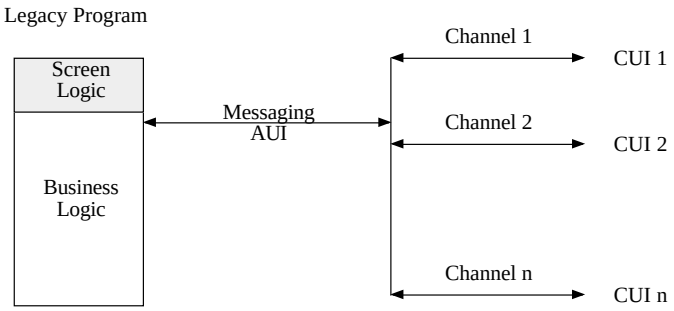


Fig. 15. Multi-channel messaging to add plasticity to a legacy system. The user interface is modelled with the AUI and the concrete user interface components communicate through a multi-channel messaging architecture. The concrete user interfaces may be automated teller machines, personal computers, telephones, etc.

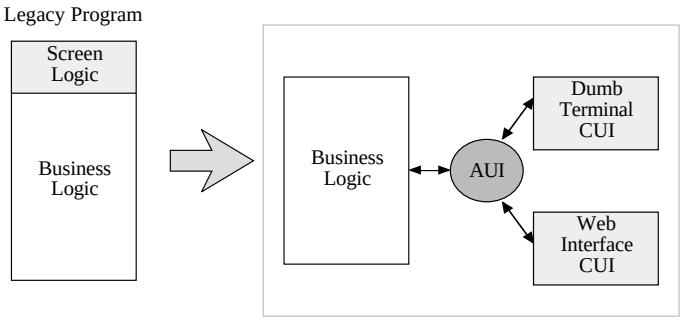


Fig. 16. Multi-session architecture for adding plasticity to a legacy system. The AUI models the interface between the user interface logic and the business logic. The AUI also manages the state necessary to achieve multi-session processing.

6 Connections

The abstract user interface notation provides a means of investigating and expressing abstract interaction. To further the research, the notation needs to be used to express a variety of interaction relations to determine common patterns and the usability of the notation. Research presented at this workshop into 'Temporal Patterns for Complex Interaction Design' will aid in identifying some of these common patterns. As well, the idea of 'Symmetry as a Connection Between Affordance and State Space' may help establish criteria for connecting CUI and AUI.

To aid in practically exercising the AUI notation a supporting user interface builder should be designed and implemented. With an AUI development environment the AUI notation could be further exercised and accessible to a larger audience. The AUI development environment will need to integrate with other user interface tools. Ideas for such integration have been presented at this workshop in the chapter on 'Tool Suite for Integrating Task and System Models Through Scenarios.'

Related to user interface plasticity is the notion that user interfaces may need to adapt to different contexts of use. The chapter on 'Task Modelling for Context-Sensitive User Interface' addresses this issue.

7 Conclusion

The abstract user interface (AUI) model and supporting notation provide a means of specifying abstract interaction to aid in the design and development of plastic interactive systems. The AUI model separates an interactive system into three components, the concrete user interface (CUI), the abstract user interface (AUI) and the computation. The CUI is concerned with issues of events and display updates, the computation is concerned with issues of application specific functionality, and the AUI relates the two.

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 CUI's for multiple platforms could be constructed for the same AUI.

By having a clearer and more structured way of expressing application interaction using the AUI notation, iterative development of interactive systems may prove to be easier and less error prone. As well, having clearly stated the relation between user interface and application, an interactive system may be more robust since the dependencies are more easily accounted for.

Temporal issues are not addressed in the AUI notation. 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. Unfortunately there may be cases where it is desirable to express temporal constraints in the AUI in terms of AUI elements. For example, some temporal constraints may need to be maintained regardless

of the given CUI or application bindings. It would be interesting to investigate an orthogonal notation based on ideas from Clock [7] for expressing temporal constraints. The AUI notation supplemented with temporal constraints would be conducive to demonstrational techniques, especially for specifying sequencing. As well, a notation based on UAN [9] for expressing the concrete user interface could provide a CUI complement to the AUI notation.

Although the 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 non-WIMP user interfaces.

Using the AUI model to improve the plasticity of existing legacy systems is an exciting area for future research. The techniques that have been used commercially need to be generalized so they may be easily applied to a range of legacy systems.

The AUI model is hoped to be 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.

Acknowledgements

The authors wish to thank the Natural Sciences and Engineering Research Council of Canada for their support. We also wish to thank T. C. N. Graham for his valuable comments and feedback.

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. AppleWorks 6, Copyright ©2000 Apple Computer. Apple Computer, 2000.
3. F. Bodart, A.-M. Hennebert, J.-M. Leheureux, and J. Vanderdonckt. A Model-Based Approach to Presentation: A Continuum from Task Analysis to Prototype. In Proceedings of 1st Eurographics Workshop on Design, Specification and Verification of Interactive Systems DSVIS'94 (Bocca di Magra, Jun 8-10, 1994). F. Paternó (ed.). Eurographics Series, Berlin, 1994, pp. 25-39.
4. T. Bray, J. Paoli, and C. M. Sperberg-McQueen, eds. Extensible Markup Language (XML) 1.0. W3C Recommendation, 1998.
5. J. Coutaz, L. Nigay and D. Salber. PAC: An Object Oriented Model for Implementing User Interfaces. ACM SIGCHI Bulletin, vol. 19, 1987, pages 37-41.
6. M. Crease, P. Gray and S. Brewster. A Toolkit of Mechanism and Context Independent Widgets. In Design, Specification and Verification of Interactive Systems (Workshop 8, ICSE 2000), Limerick, Ireland, 2000, pp. 127-141.
7. T. C. N. Graham. Declarative Development of Interactive Systems. Volume 243 of Breichte der GMD. R. Oldenbourg Verlag, July 1995.

8. T. C. N. Graham and T. Urnes. Integrating Support for Temporal Media into an Architecture for Graphical User Interfaces. In Proceedings of the International Conference on Software Engineering (ICSE97). IEEE Computer Society Press, Boston, USA, May 1997.
9. H. R. Hartson, A. C. Siochi and D. Hix. The UAN: A User-Oriented Representation for Direct Manipulation Interface Designs. ACM Transactions on Information Systems, 1990, 8(3):181-203.
10. 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.
11. R. D. Hill. The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications. In Human Factors in Computing Systems (Monterey, California, USA), 1992, pages 335-342.
12. P. Hudak, S. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, 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.
13. M. P. Jones. An Introduction to Gofer. Functional programming environment, Yale University, 1991.
14. B. A. Myers. A New Model for Handling Input. ACM Transactions on Information Systems, 1990, 8(3):289-320.
15. G. E. Pfaff, editor. User Interface Management Systems. Springer-Verlag, Berlin, November 1983.
16. 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.
17. 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.
18. The UIMS Tool Developers Workshop. A Metamodel for the Runtime Architecture of an Interactive System. SIGCHI Bulletin. Volume 24, Number 1, 1992, pages 32-37.

Appendix. AUI Syntax

auiSpecification ::= { equation }
equation ::= functionDef | functionSignature | typeDef | typeSynonyn | matchEqn
functionDef ::= functionName { pattern } = expression [whereClause]
whereClause ::= **where** { equation } **end where**
functionSignature ::= functionName **::** typeName1 { -> typeName2}
typeDef ::= typeName { typeVar } ::= constructorExpr { | constructorExpr }
constructorExpr ::= constructor { typeExpr }
typeSynonym ::= typeName == typeExpr
matchEqn ::= **match** identifier = pattern
expression ::= identifier | literal | functionName { expression } |
unaryOperator expression |

expression binaryOperator expression | choose | gel |
expression , **if** expression | comprehension | **(** expression **)**

list ::= [[expression { , expression }]]
set ::= **{** [expression { , expression }] **}**
tuple ::= < [expression] { , expression } >
unaryOperator ::= **-** | **not** | **first** | **rest**
binaryOperator ::= arithmeticOps | logicOps | listOps | functionComposition
arithmeticOps ::= **+** | **-** | **/** | ***** | ^ | **div** | **rem**
logicOps ::= **==** | **<=** | **>=** | **=** | **<** | **>** | **and** | **or**
listOps ::= **:** | **++**
functionComposition ::= **.**
comprehension ::= listComprehension | setComprehension
listComprehension ::= [expression | generator { , generator }]
setComprehension ::= **{** expression | generator { , generator } **}**
generator ::= name **<-** listOrSetExpression |
< name , name > **<-** 2-tupleListOrSetExpression |
< name , name , name > **<-** 3-tupleListOrSetExpression | ...
listOrSetExpression ::= listExpression | setExpression
listExpression ::= [number **..** number] | [char **..** char] | [type]
setExpression ::= **{** number **..** number **}** | **{** char **..** char **}** | **{** type **}**
pattern ::= identifier | literal | _ |
< [pattern { , pattern }] > | [[pattern { , pattern }]] |
{ [pattern { , pattern }] **}** | **(** [pattern { , pattern }] **)** |
pattern **:** pattern | constructor { pattern } | **(** pattern **)**
gel ::= **canvas** { attribute } **{** [pin { , pin }] **}** | shape { attribute }
shape ::= **circle** | **line** | **box** | **point** | **label**
pin ::= < gel , xy-offset > | gel xy-offset
xy-offset ::= < number , number > | **X** number **Y** number
attribute ::= < number , number > | **(Fill** fill **)** | **(Arrows** arrows **)** |
(Font font **)** | ...
choose ::= **choose** [name] [cardinality] setOrList
cardinality ::= ***** | number | min **..** max | min **.. ***
externalFunction ::= **external**_functionName | **ext**_functionName | _functionName

Note: *functionName*, *typeName*, *typeVar*, *typeExpr* and *name* are all identifiers.