FlexClock, a Plastic Clock Written in Oz with the QTk toolkit

Donatien Grolaux[1], Jean Vanderdonckt[2], Peter Van Roy[1]
Université catholique de Louvain
[1]Department of Computing Science and Engineering (INGI) – [2]School of Management (IAG)
[1]Place Sainte Barbe, 2 – [2]Place des Doyens, 1
B-1348 Louvain-la-Neuve, Belgium
+32-10/47.{2415, 8374, 8525}
ned@info.ucl.ac.be, vanderdonckt@isys.ucl.ac.be, pvr@info.ucl.ac.be

ABSTRACT
This paper focuses on the techniques involved in building an interactive application using a plastic user interface. These techniques take advantage of the QTk toolkit, a toolkit that features innovative and powerful concepts when compared to more classical object-oriented toolkits. These features are possible thanks to the underlying programming language used, Oz, and in particular:

a. its support to symbolic records that can express data structures in a similar yet more complete way than XML.

b. its capacity to wrap any languages entities into higher order data structures.

This paper shows by a case study how the combination of QTk and Oz helps developers write plastic user interfaces very easily.

Keywords
Model-based user-interfaces, reengineering, knowledge bases, web sites, presentation model.

INTRODUCTION
Recent years have seen the introduction of many types of access devices including PDA and mobile phones [7,16]. These devices created new needs for the user interfaces: smaller display, no keyboards or only a few keys available, different pointing devices. At first, the applications running on these devices used specific techniques to manage the user interface. As the CPU power of these devices is now increasing, it is more and more to the advantage of the user to have a single application running on different devices. These applications must adapt their user interface according to the specification of the device it's running on.

In general, applications can often exploit context changes to offer plastic user interfaces for the advantage of the end-user [22]. A very common case is the variation of the size of the window of an application. When the user is resizing the window, all applications reorganize their widgets to adapt to the new situation. However this is often limited to displaying less information when the size is reduced, and use scrollbars to let the user access the hidden information. A plastic user interface could instead select different views to display part of the information displayed according to the size of the window. The usability of the application is enhanced: the link between the size of the window and the information available to the user is reduced.

While developing graphical user interfaces can be hard, developing plastic user interfaces can be even more complex to achieve: it involves a multiplicity of UI for manipulating the same set of data from the application, and the corresponding links between these UI and the functions belonging to the semantic core of the interactive application. The intertwining of these different aspects makes the software architecture modularization very demanding. Not only in terms of independence of UI with respect to the semantic core, but also in terms an autonomous presentation with respect to the dialog part in the UI, and the definition of rules that switch from one presentation to another depending on variations of the context of use (here, restricted as windows resizing).

FLEXCLOCK ARCHITECTURE
The FlexClock application [8] is a replacement to the well-known clock utility of Unix systems. FlexClock displays the current time and date in several different views according to the size of the window

FlexClock defines different display for a clock, from a very simple HH:MM display to a complex display composed of an analog clock, a digital clock and a monthly calendar. There are 17 different possible displays defined. They all exist simultaneously inside the application, and there is no difference between the display currently visible to the user and the other displays.

Views are characterized by 3 points:

1. What they display. In the case of the toolkit in use (QTk), this corresponds to a data structure semantically equivalent to XML (Oz records) defining the initial state of the display, ie the widgets and their geometry. This data structure is part of the Oz language itself and is transparently integrated with the remaining of the code of the application.
2. How to update the displays. Each display defines this by a one-argument procedure taking the time as parameter.
3. How to choose between displays. Each display defines an information containing the minimal width and height in pixel size required to display it. This information is enough to select the potential candidates for a given available size among all the displays.

The selection rule uses also the minimal width and height information to determine a 'best' candidate, the one that is actually made visible to the user. Let wh and ww be the height and width in pixels of the window, and mh_i and mw_i the minimal height and width in pixels of the display i. The chosen display is the one:

a. That can be displayed: $mh_i > wh$ and $mw_i > ww$.
b. That minimizes $(wh-mh_i)^2+(ww-mw_i)^2$ (In the case of a tie, just pick a random one).

Note that each display defines a minimum size, but not a maximum one: each display can be stretched continuously and the widgets configure themselves according to the geometry management used to define them. The text widgets displaying the digital clocks will just center themselves inside the space that is made available to them, while the analog clock will stretch as much as possible to display the biggest clock possible. As a result, if the user changes the size of the FlexClock window in such a way that it is always the same display that is visible, this display will itself take advantage of the size available as much as possible. FlexClock incorporates a set of 17 different presentations recorded as views, some of them being reproduced in Fig. 1.

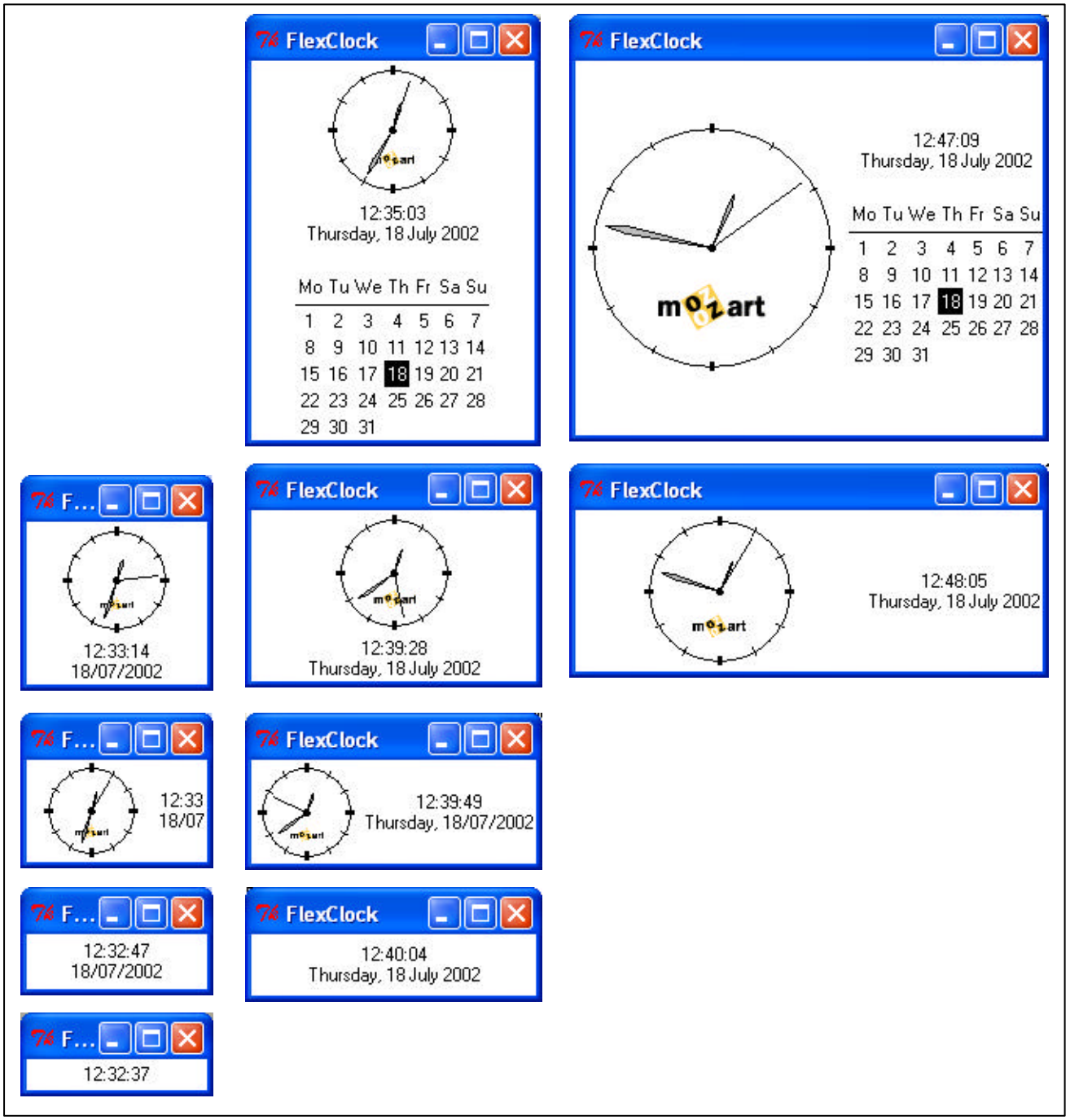

Figure 1. Some possible views as displayed by FlexClock.

The 17 displays are defined in a list, each one taking 3 to 6 lines of code depending on the complexity. The subsequent lines of code build the actual widgets from these definitions, and register their update procedure. Then, there is a need to define a thread that calls all registered update procedures each second with the new time.

The Place procedure defines the selection rule that chooses between the displays, according to the current window size. The toolkit provides a Configure event that is triggered each time the widow is resized. This event is set up to call the Place procedure. The remaining lines of codes define the custom widgets used by FlexClock (analog clock and calendar), and formatting properties for the digital clock.

This application is conceptually simple:

- FlexClock contains several different views.
- Each view defines a unique way of displaying the current time and date.
- Each second, all views are updated.
- Each time the size of the window changes, the best possible view is selected. Each view is characterized by a minimum width and height below which it can't be displayed. The selection rule of FlexClock is to display the view that minimizes the norm 2 difference between the width and height of the window and the minimum width and height of the view. Other selection rules can be used, but the norm 2 gives good instinctive results.

Using the Oz language [6,10,13] and the QTk toolkit [9], this application is simple to implement: there is a direct mapping between the conceptual view of the application and its implementation.

- Each view is defined in a single Oz data structure
- Using an event-based mechanism, each view defines how it must be updated every second
- Using an event-based mechanism, each time the user changes the size of the window, a selection rule is applied and changes the displayed view if necessary.

Oz and QTk

The points of Oz (http://www.mozart-oz.org) and QTk that are of particular importance here are [13,20]:

- Oz symbolic records. Records are characterized by a symbolic label and by a set of symbolic features to which corresponds Oz values [20].

```
label(text:"Ok" background:red)
```

  Semantically, Oz records cover XML, and as such can be used to express user interfaces declaratively. However, as records are native data structures of Oz, there is a complete support from the language to manipulate them. Also the link between the (static) declaration of the user interface and the (dynamic) remaining of the application is natural by specifying an Oz variable. For example:

```
label(text:"Ok" background:red handle:MyLabel)
```

  In this example, the Oz MyLabel variable can be used to manipulate the label widget, for example to change the displayed text of the label:

```
{MyLabel set(text:"Cancel")}
```

  The QTk toolkit can build a window from a correctly formatted Oz record. By consequent, each different FlexClock view is defined by a single Oz record.

- Higher order Oz wrappers. Procedures are first class entities of the Oz language, and as such can be stored in variables or used as parameters of other procedures. For each different view, the update of the time of the view is wrapped into a procedure taking the time as parameter.

Definition of possible presentations

Windows contents change according to the size of the window as it is currently being manipulated. For each window, the height and the length uniquely determine its presentation (Fig. 2 to Fig.4).

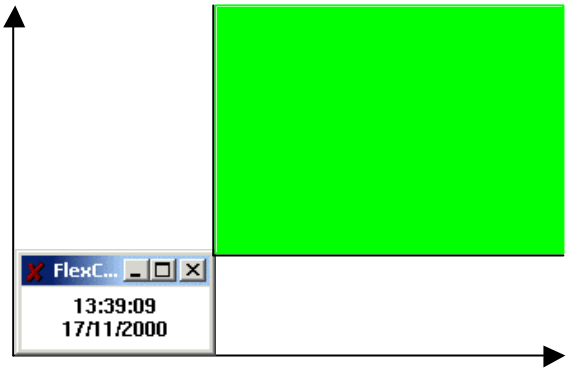

Figure 2. The plain green area represents the visible range of this single view.

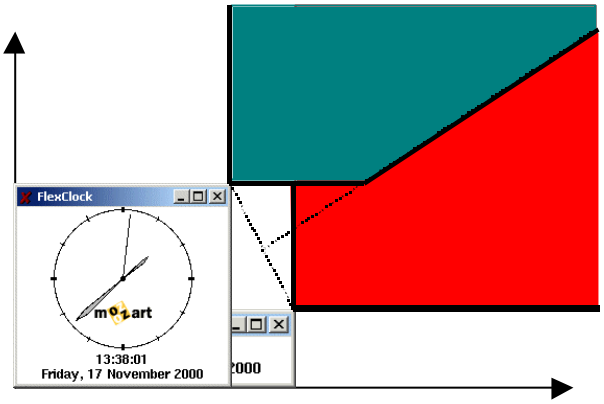

Figure 3. Plain colors represent the visible range of these two views.

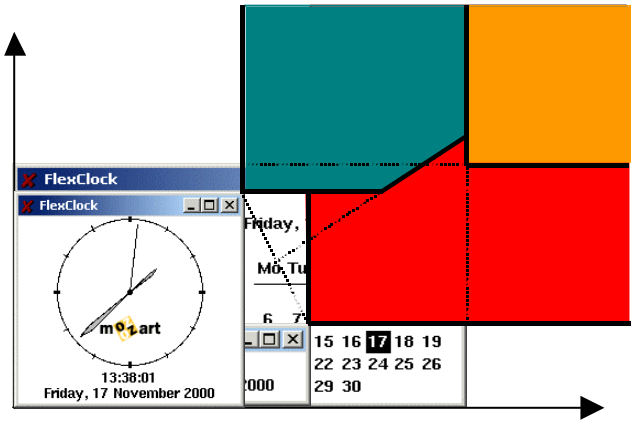

Figure 4. Plain colors represent the visible range of these three views.

Figures 2 to 4 show how it is possible to divide up the plane among the views. The procedure Place does the calculation and displays the best view in the placeholder:

```
proc{Place}
  WW={QTk.wInfo width(P)}
  WH={QTk.wInfo height(P)}
  fun{Select Views Max#CH}
    case Views
    of W#H#Handle|R then
      This=(W-WW)*(W-WW)+(H-WH)*(H-WH)
    in
      if W<WW andthen H<WH andthen
        (Max==inf orelse This<Max) then
        {Select R This#Handle}
      else
        {Select R Max#CH}
      end
    else CH end
  end
in
  {P set({Select Views inf#Views.1.3})}
```

This starts with a minimum of inf, representing infinity, and is reduced for each view with a smaller distance that is small enough to be displayed. When the window is resized, Place has to be called to set the correct view according to the new size of the window. This is done by binding the <Configure> event of QTk:

```
{P bind(event:´<Configure>´
        action:Place)}
```

At any time, each window resizing triggers the identification of the closest possible configuration depending on the definition of existing views (Fig. 2-4). As many competitive candidates can be identified, there is a need to identify one and only one possible presentation. The chosen display is the one:

a. That can be displayed: $mh_i > wh$ and $mw_i > ww$.

b. That minimizes $(wh-mh_i)^2+(ww-mw_i)^2$ (In the case of a tie, just pick a random one).

where (wh,ww) are the coordinates of the upper right corner of the window on the plane and (mh_i,mw_i) are the coordinates of the upper right corner of the respective candidate regions. (mh_i,mw_i) is also the minimal size required by the window to display each view i. 17 displays are currently defined in FlexClock. There is a minimum threshold where to window is not high or not large enough to display any content possible. This region is represented in Fig. 5.

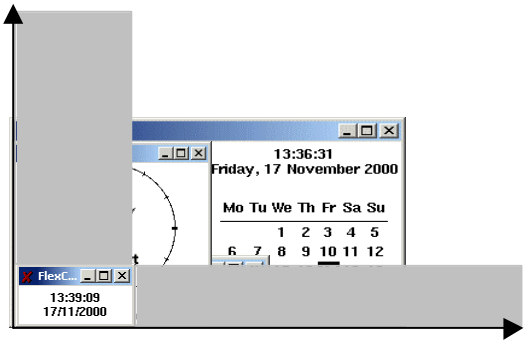

Figure 5. The gray areas represent window areas where there is no display possible.

Several other selection rules can be used to choose the current display. The FlexClock one minimizes the distance between the upper right corner of the window and the upper right corners of the possible candidates, ie it minimizes the 2-Norm distance between these points. The 1-Norm or the ∞-Norm could be also used, however instinctively they give less interesting results. Of these three norms, the 2-Norm is the one that best fulfills the whole area visible to the user, and we believe the result is very natural to the users.

FLEXCLOCK IMPLEMENTATION

A FlexClock view is characterized by 3 things:

1. The Oz record defining the user interface of the window.
2. A time update procedure.
3. A minimum width and height expressed in pixels.

As an example, here is the definition of the simplest view:

```
r(desc:label(handle:H1 glue:nswe bg:white)
  refresh:proc{$ T} {H1 set(text:{FmtTime T})} end
  area:40#10)
```

There are two orthogonal events to manage:

1. Time update. Every second, the update procedure of all views are called with the new time as parameter.
2. Window size change. Each time the user changes the size of the window, the selection rule is applied and the selected view is displayed.

These two kinds of events are managed in separate parts of the text of the code of the application, respecting the separation of concern principle.

The declarations of the 17 different views takes around 100 hundred lines of code. The complete code of the application also contains the definition of the analog clock and the calendar widgets, the management of the Mozart icon and the selection rule to select the view to display. This takes around 400 lines of code.

IMPLEMENTATION OF PLASTICITY

The FlexClock is a good example of implementing a plastic user interface with dedicated capabilities. In the Oz/QTk code of the FlexClock, it is easy to identify the different modules that directly contribute to the plasticity property. Plasticity is the "capacity of an interactive system to *withstand variations* of *contexts of use* while *preserving usability*" [3,21,22]. The usability of an interactive system is evaluated against a set of properties selected in the early phases of the development process. A user interface preserves usability if the properties elicited at the design stage are kept within a predefined range of values as adaptation occurs to different contexts of use.

In [4], Calvary *et al.* characterize the different modules that participate to a working plastic user interface thanks to a mechanism called the *software probe*. These modules are graphically represented in fig. 6 and are revisited below.

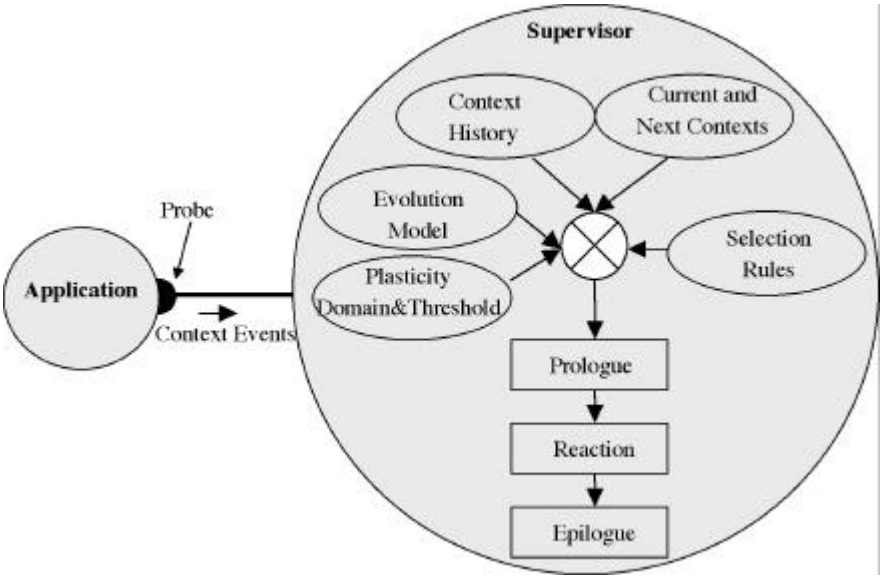

Figure 6. The adaptation process to context changes based on a transparent probe mechanism (from [4]).

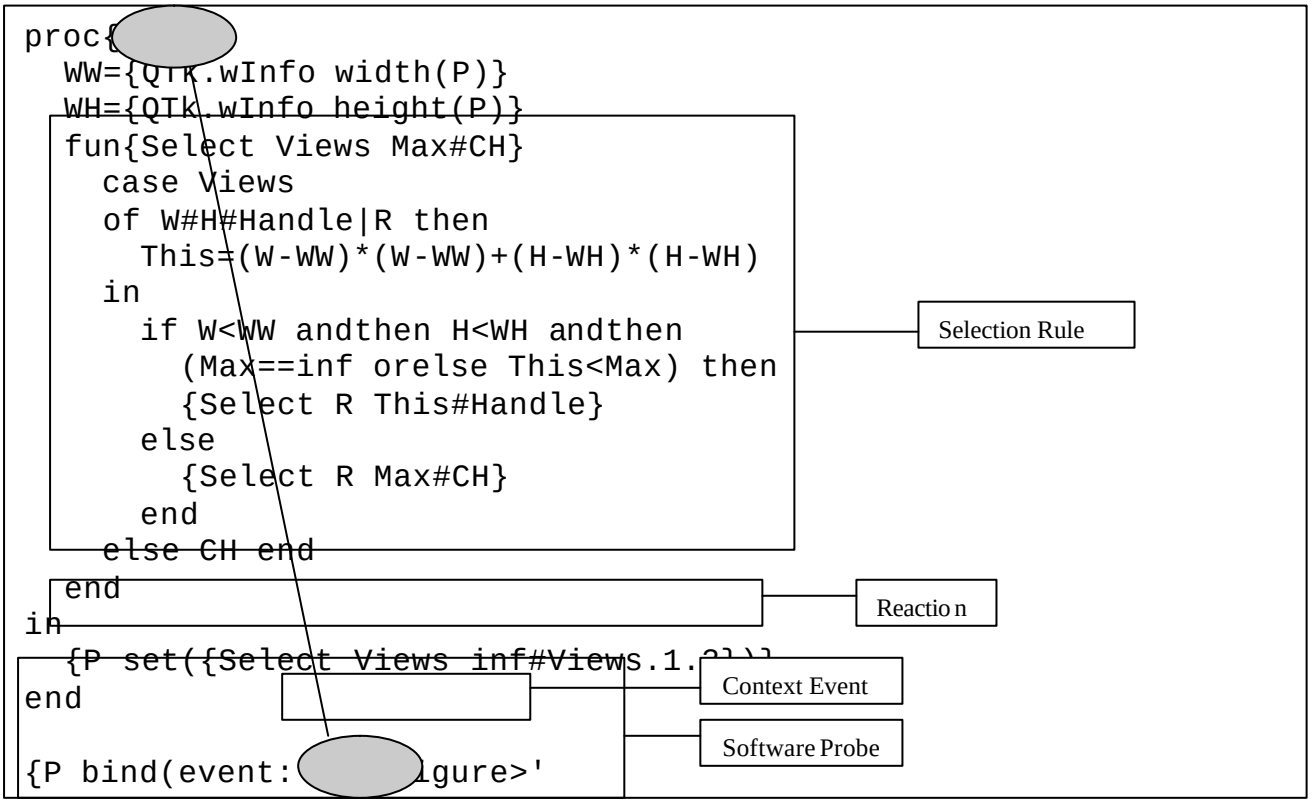

Figure 7. Implementation of the adaptation process in the FlexClock code.

The different modules of fig. 6 are [3,4,21,22]:

- The *Software probe* itself: it is a software mechanism that is responsible to detect any change of the context of use by sensing the application. The probe is then responsible to check whether the actual user's behavior conforms to the designers' expectation. For doing so, designers specify *Situation ® Reaction* rules that describe users' expected behav-

iors. In the case of FlexClock, there is no genuine probe mechanism placed over an already existing application since there is no functional core and sensing the context is here summarised to the detection of any change in the window size, cfr fig. 7.

- The *Context events*: they characterise at a high level of abstraction any significant change of the context of use. FlexClock is only interested in the window resize event, denoted as <Configure> in fig. 7.
- The *Current and Next Contexts*: they respectively capture the current context of use and the one that will be produced after a detection of context change. In FlexClock, only the coordinates of the newly resized window are captured and communicated to select an appropriate configuration. Each time a window resize occurs, the computing of a possibly new configuration is triggered.
- The *Selection rules* and the (*Prologue, Reaction, Epilogue*) are responsible for identifying possible reactions to a change of context, for selecting one which is appropriate and for applying it. The supervisor executes the selected prologue, the reaction and the epilogue. In FlexClock, instead of using a single display at a time with a transition (Prologue, Reaction, Epilogue) mechanism point of view, it uses a simultaneous multiple-display point of view, which is directly reflected down to the text of the code of the application. The selection rule is defined inside the Place procedure, cfr fig. 7. Letting the system spend time and resources managing displays that are not visible to the user has not been considered due to the sake of effectiveness and performance. In UI that are more complex than FlexClock, a transition approach is likely to be used instead. However, it may be easier to think about the application in term of the multiple-display point of view than from the transition point of view. Depending on the situation, the application developers can take advantage from using both point of views, and as such, the transition approach should not be considered as the only valid one.
- The *History mechanism*: it records context transitions along with their migration costs as the user runs the system across multiple sessions. It also maintains a list of previously used configurations for previously encountered contexts. There is no notion of context history in FlexClock as there is no need to keep the previous states. Indeed, a new configuration can be computed at any time.
- The *Evolution Model*: it is responsible for specifying reactions to context changes. In the case of FlexClock, the evolution model no longer consists of a model, but a sequence of operations that remain constant for each change: make the selected display visible and hide all others, cfr fig. 7.
- The *Plasticity domain & threshold*: the plasticity domain of a user interface is the surface formed by all couples (platform, environment) the user interface is able to handle. The boundaries of this surface defines the plasticity threshold of the user interface. In FlexClock, the plasticity domain & threshold should specify when a view becomes invalid and should be replaced by another one, the selection rules choose the new view and the evolution model specifies the transition between the old and new view. The plasticity domain is defined by the sum of minimum width and height of the different possible views. Consequently, the plasticity domain is the entire plan where the minimal upper left rectangle is removed (represented in fig. 5).

In summary, the plasticity FlexClock architecture can be graphically depicted by a series of displays as represented in fig. 8.

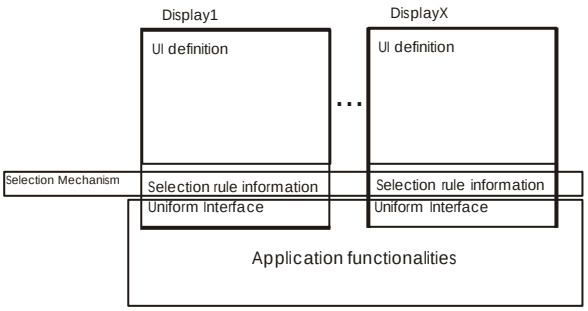

Figure 8. Plasticity architecture in FlexClock.

RELATED WORK

QTk's main virtue is its simplicity: declarative parts of the user interface are programmed in a declarative way, while dynamic parts are programmed in a classical object-based imperative way; both parts are still expressed in the same programming language. The Mozart system (http://www.mozart-oz.org) is the current implementation of the Oz language. QTk is part of this system and is being used for several actual applications.

QTk can be compared to several other works in the domain of UI development environments that support designers and developers in designing UIs according to a model-based approach leading to executable plastic UIs. We now compare QTk with some selected work regarding to the executability of models and regarding to other model-based approach [15]. Other works are for instance [5,7,18,19].

Jacob [11] is one of the pioneers who introduced the need for executable UIs from their specifications and demonstrated its feasibility, in this case from a state-transition diagram expressing the dialogue. Lean Cuisine+ [17] is an executable semi-formal graphical notation for specifying the underlying behavior of event-based direct manipulation interfaces. It is a multi-layered notation which supports the early design phase of the interface development life cycle.

In [1], Bomsdorf & Szwillus developed a task model enriched with complex relations between tasks that can be executed. Depending on the abstraction level of the development process, graphical representations or early ideas of screen layout can be attached to it. With this technique, prototypes can be used very early in the design process, improving the capabilities to evaluate the model.

In these examples, various models (a dialogue model for Jacob and LeanCuisine, a task model for Bomsdorf and Szwillus) are exploited to obtain an executable UI. In QTk, presentation and dialogue models are considered as the terminal models needed to run a UI. But other models can be incorporated and translated into Oz records when needed and depending on the needs of the target system. For instance, the domain-to-presentation mapping problem [7] can be solved by writing selection rules and arrangement rules that map Oz data structures (i.e. the domain model) to widget structures (i.e. the presentation model), along with their predetermined behavior (i.e. the dialogue model). Any other mapping can be equally established.

Tcl is a simple scripting language for controlling and extending interactive applications: the Tcl interpreter is designed to be easily extended with application specific commands. Tk extends Tcl with command for building user interfaces. Tcl/Tk [14] provided us with at least two advantages: (i) the high level of abstraction of Tcl/Tk makes it easy to generate implementations, and (ii) its free availability for multiple computing platforms, such as OSF/Motif and Microsoft Windows. QTk acts as a front-end to Tcl/Tk. QTk is an interpreter that maps declarations and direct manipulations from the Oz language to Tk commands.

Bumbulis *et al.* [2] built a methodology and a system allowing the designer to describe a user interface with an IL specification language that results into Tcl/Tk instructions. In some sense, specifications contained in a presentation model and a dialogue model are examined and analyzed to produce an executable UI. However, the IL specification language differs from the Tcl scripting language, thus forcing developers to constantly switch from one language to another. QTk is very integrated with the Oz language, providing Oz only abstractions. Consequently, there is no need to switch between Oz and Tcl/Tk, in fact there is even no need to known Tcl/Tk at all.

Miyashita *et al.* [12] argued that programming by visual example can be achieved through their TRIP3 tool. In this system, a bi-directional translation exists between the application data contained in a domain model and the pictorial data contained in a presentation model. This system is able to generate mapping rules from the domain model to the presentation model for the translation from example data and its corresponding example presentation. Note that QTk is advantageous over TRIP3 in that multiple mappings between several models can be specified at once, whereas TRIP3 has only one type of mapping. On the other hand, TRIP3 and QTk both share the scheme in which the presentation model can be expressed in a declarative manner that can be instantly executed.

CONCLUSION

When developing plastic user interfaces, a number of factors can help developing the application:

- Efficiency of the toolkit itself. User interfaces are more easily built from a declarative data structure than from a succession of instructions. Languages that support such data structures natively allow easy manipulations of such data structures. Also the gate between the declaration of the UI and the functional core of the application is trivial (a variable does the work), instead of an artificial external link.
- Capability of the language to wrap complex entities. Easy development of plastic user interfaces is possible if and only if it is no more expensive than non-plastic user interfaces. This can be achieved if the plastic part of the UI is wrapped in a single language entity that is considered non-plastic from the rest of the application.
- Application of the separation of concern principles, and event-based techniques.

Oz and QTk are strong with all these points, and it shows when using them to develop true plastic applications. Thanks to the interpretation of the code, the code remains the same for all computing platforms supported by the Mozart system, thus leading to different presentations depending on the native Look & Feel of the computing platform (e.g., Unix in Fig. 9).

Figure 9. FlexClock in other computing platforms.

APPENDIX.

The FlexClock code, implementation description and demonstration have been developed by Donatien Grolaux in the context of the PIRATES Project and are available from the following web site:

http://www.info.ucl.ac.be/people/ned/FlexClock/index.html

REFERENCES

1. Bomsdorf, B., Szwillus, G., *Early Prototyping Based on Executable Task Models*, in Proc. of ACM Conference on Human Factors in Computing Systems CHI'96 (Vancouver, April 14-18, 1996), ACM Press, New York, 1996, pp. 254–255.
2. Bumbulis, P., Alencar, P.S.C., Cowan, D.D., Lucena, C.J.P., *Combining Formal Techniques and Prototyping in User Interface Construction and Verification*, in Palanque, Ph;, Bastide, R. (eds.), Proceedings of 2nd Eurographics Workshop on Design, Specification and Verification of Interactive Systems DSV-IS'95 (Bonas, June 7-9, 1995), Springer-Verlag, Vienna, 1995, pp.174–192.
3. Calvary, G., Coutaz, J., Thevenin, D., *A Unifying Reference Framework for the Development of Plastic User Interfaces*, Proceedings of 8th IFIP International Conference on Engineering for Human-Computer Interaction EHCI'2001 (Toronto, 11-13 May 2001), R. Little and L. Nigay (eds.), Lecture Notes in Computer Science, Vol. 2254, Springer-Verlag, Berlin, 2001, pp. 173-192.
4. Calvary, G., Coutaz, J., Thevenin, D., *Supporting Context Changes for Plastic User Interfaces: a Process and a Mechanism*, in "People and Computers XV – Interaction without Frontiers", Joint Proceedings of AFIHM-BCS Conference on Human-Computer Interaction IHM-HCI'2001 (Lille, 10-14 September 2001), A. Blandford, J. Vanderdonckt, and Ph. Gray (eds.), Vol. I, Springer-Verlag, London, 2001, pp. 349-363.
5. Crease, M., Gray, Ph., Brewster, S., *A Toolkit of Mechanism and Context Indepndent Widgets*, in Proceedings of 6th International Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS'2000 (Limerick, June 2000), pp. 127-141.
6. Duchier, D., Kornstaedt, L., Schulte, Ch.: The Oz Base Environment. (February 7, 2000). Accessible at http://www.mozart - oz.org/documentation/base/index.html
7. Eisenstein, J., Vanderdonckt, J., Puerta, A., *Model-Based User-Interface Development Techniques for Mobile Computing*, in Proceedings of ACM Int. Conf. on Intelligent User Interfaces IUI'2001 (Santa Fe, January 14-17, 2001), J. Lester (ed.), ACM Press, New York, 2001, pp. 69-76.
8. Grolaux, D., Van Roy, P., Vanderdonckt, J., *QTk: An Integrated Model-Based Approach to Designing Executable User Interfaces*, in PreProc. of 8th Int. Workshop on Design, Specification, Verification of Interactive Systems DSV-IS'2001 (Glasgow, June 13-15, 2001), Ch. Johnson (ed.), GIST Tech. Report G-2001-1, Dept. of Computer Science, Univ. of Glasgow, Scotland, 2001, pp. 77-91. Accessible at http://www. dcs.gla.ac.uk/~johnson/papers/dsvis_2001/grolaux.
9. Grolaux, D., *QTk Module*, March 13, 2000. Accessible at http://www.info.ucl.ac.be/people/ned/QTk/http-html/index.html
10. Haridi, S., Franzén, N.: Tutorial of Oz. (February 7, 2000). Accessible at http://www.mozart-oz.org/documentation/tuto rial/index.html
11. Jacob, R.J.K., *An Executable Specification Technique for Describing Human-Computer Interaction*, chapter 8, in R. Hartson (ed.), "Directions in Human-Computer Interaction". Ablex Publishing Company, 1987, pp. 211–242.
12. Miyashita, K., Matsuoka, S., Takahashi, S., *Declarative Programming of Graphical Interfaces by Visual Examples*, in Proc. of ACM Conf. on User Interface Software Technology UIST'92 (Pittsburgh, November 15-18, 1992), ACM Press, New York, 1992, pp. 107–116.
13. Mozart Consortium, The Mozart Programming System (Oz 3). (January 1999). Accessible at http://www.mozart-oz.org
14. Ousterhout, J.K., *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, 1994.
15. Paternò, F., *Model-based Design and Evaluation of Interactive Applications*, Springer Verlag, Berlin, 1999.
16. Paternò, F., Santoro, C., *One Model, Many Interfaces*, in Proceedings of 4th International Conf. on Computer-Aided Design of User Interfaces CADUI'2002 (Valenciennes, 15-17 May 2002), Kluwer Academics Pub., Dordrecht, 2002, pp. 143-154.
17. Phillips, Ch., *Serving Lean Cuisine+: Towards a Support Environment*, in Proceedings of, the CHISIG Annual Conference on Human-Computer Interaction OZCHI'94 (Melbourne, November 28-December 1, 1994), Ergonomics Society of Australia, Canberra, 1994, pp. 41–46.
18. Salber, D., Dey, A.K., Abowd, G. D., *The Context Toolkit: Aiding the Development of Context-Enabled Applications*, in Proceedings of ACM Conference on Human Factors in Computing Systems CHI'99 (Pittsburgh, 15-20 May 1999), ACM Press, New York, 1999, pp. 434-441.
19. Schneider, K.A., Cordy, J.R., *Abstract User Interfaces: A Model and Notation to Support Plasticity in Interactive Systems*, in Johnson, Ch. (ed.), Proceedings of 8th International Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS'2001 (Glasgow, June 13-15, 2001), Lecture Notes in Computer Science, Vol. 2220, Springer-Verlag, Berlin, 2001, pp. 28-48.

20. Smolka, G., *The Oz Programming Model*, in Lecture Notes in Computer Science, vol. 1000. Springer-Verlag, Berlin, 1995.
21. Thevenin, D., Coutaz, J., *Plasticity of User Interfaces: Framework and Research Agenda*, in Proc. of 7th IFIP International Conference on Human-Computer Interaction Interact'99 (Edinburgh, August 30 - September 3, 1999), Chapman & Hall, London, pp. 110-117.
22. Thevenin, D., *Adaptation en Interaction Homme-Machine: Le cas de la Plasticité*, Ph.D. thesis, Université Joseph Fourier, Grenoble, 21 December 2001.