QTk: An Integrated Model-Based Approach to Designing Executable User Interfaces

Donatien Grolaux[1], Peter Van Roy[1], and Jean Vanderdonckt[2]

Université catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium
[1]Département d'Ingénierie Informatique, Place Sainte Barbe, 2
{ned,pvr}@info.ucl.ac.be
[2] Institut d'Administration et de Gestion, Place des Doyens, 1
vanderdonckt@qant.ucl.ac.be

Abstract. QTk is a tool built on top of Tcl/Tk that allows user interface designers to adopt a cost-effective model-based approach for designing executable user interfaces. QTk is based on a descriptive approach that uses a declarative style where appropriate (symbolic records to specify widget types, initial states, and geometry management; procedure values to specify actions), augmented with objects and threads to handle the active part of the interface. QTk offers four original advantages: unicity of language (only one language serves as both modeling and programming language), reduced development cost (the interface model immediately gives rise to an executable user interface), tight integration of tools (specification, construction, and execution tools are all integrated), and improved expressiveness (the interface model is very compact to produce and cheap to manipulate). The advantages are made possible by a tight integration with a multiparadigm programming language, Oz, that supports symbolic data structures, a functional programming style, an object programming style, and cheap threads. QTk is a module of the Mozart Programming System, which implements Oz. We show how to port QTk to Java, which allows to retain some but not all of the tool's advantages.

1 Introduction

The aim of exploiting a model-based approach [14] for designing user interfaces (UIs) consists in capturing UI specifications into models from which a running UI can be derived. It is useful to be able to specify a desired UI formally into models before building it, particularly if a mockup suitable for testing can be obtained directly from the specification [1,9,10]. Each such model has three properties [16,18]:

1. *Declarative*: specifications contained in each model should be expressed in a declarative specification language called *interface modeling language*.
2. *Editable*: specifications can be edited either manually by a human operator or automatically by an automaton.
3. *Analyzable*: specifications are expressive enough to be processed in order to perform analysis operations. Results of this analysis can be in turn used for pro-

ceeding with the next steps of the UI design process.

Models involved in a model-based approach typically include [2,4]: a task model, a user model, a domain model, a platform model, a presentation model, and a dialogue model. The last two models are indispensable to obtain a finally executable UI. To obtain such a UI, specifications contained in models can be manipulated in:

- A 'generate-compile-execute' cycle: specifications are processed at design time by software to automatically generate the UI code, possibly with some design assistance. This code is usually written in the programming language of a target development environment or in terms of instructions available in a User Interface Management Systems (UIMS) [6]. It is then compiled and executed to get the running UI. Any model modification results in a new cycle.
- A 'specify-interpret' cycle: specifications are progressively refined so as to create a subset that is expressive enough to be interpreted at any time (design time or execution time). When this subset reaches a reasonably advanced status, the specifications are released and interpreted again to get the running UI. The model can be modified at any time.

These cycles received limited success due to several constraints, such as:

1. The multiplicity of languages: although some intelligent model editor may free designers from seeing the interface modeling language, this language is still different from the programming language, thus introducing many discrepancies and difficulties in generating programming code from the specifications or interpreting them. This complexity of these generation and interpretation tasks is very high.
2. The development cost: developing a generator or an interpreter requires a high development effort that is not cost-effective. This cost can even be multiplied if a generator is needed for several programming languages, for example from a 'X' modeling language to HTML, Java, C++, and Visual Basic.
3. The poor integration of tools: five broad categories of tools are identified [6]: *requirements tools* (that are used by requirements specialists to formulate requirements); *specification tools* (that are used by system designers to produce specifications); *construction tools* (that are used by implementers to transform specifications into coded modules); *execution tools* (that are used by system administrators to assemble and bind modules into interactive systems); and, *evaluation tools* (that are used by evaluators). In most model-based approaches, most of these tools are needed, but are still separated: no smooth transition can be provided between them.
4. The reduced expressiveness of modeling language: the success of a model-based approach highly depends on the expressiveness of its interface modeling language. Some approaches have proven reasonably effective in narrow domains since the language is dedicated. However, no modeling language has been shown to be applicable at a sufficiently general level. Additionally, the compactness of this language is also a concern: modeling languages tend to be verbose and their translation into programming language is even more verbose and hard to understand.

In this paper, we address the above shortcomings by introducing QTk, a develop-

ment module built on top of the Oz language that allows designers to quickly develop executable UIs in the same integrated environment according to a model-based approach. The remainder of this paper is structured as follows: the next section provides an introduction of the Oz programming language. Section 3 details QTk and its underlying model-based approach to designing executable UIs. Section 4 describes the QTk implementation. Section 5 compares QTk with related work, with respect to other tools providing executable UIs and model-based approaches to designing UIs. Section 6 concludes the paper by highlighting four salient properties of QTk that address the above shortcomings (unicity of language, reduced development cost, tight integration of tools, and improved expressiveness) and by introducing some future work.

2 The Oz Programming Language

Oz [12,17] is a multiparadigm programming language with high-level support for symbolic data structures including lists and records. These records are very well suited for a model-based approach for designing UIs as model elements can be easily represented with these structures. QTk is an Oz module implementing a function building UIs from a correctly formatted Oz record, using a declarative approach that is also editable and analyzable.

As records are natural data structures of the language, it is easy to dynamically create or manipulate them in the functional paradigm provided in Oz. The UI specification being itself an Oz record, the dynamic creation of custom UI becomes as easy as dynamic manipulation of records. Oz is a dynamically-typed language, i.e., types are checked at runtime, which allows for great flexibility in the kinds of record manipulations. The UI description is a part of the language itself and can directly have references to live entities of the application. The interpretation of the specification can be done on-the-fly at any point of the execution, making dynamic creation of UIs very easy to implement.

2.1 Oz Data Structures

This section will detail some of the many data structures in Oz, showing useful properties for the remainder of this article.

2.1.1 Atom

An *atom* is a symbolic constant that has a printable representation made up of a sequence of alphanumeric characters starting with a lower case letter, or arbitrary printable characters enclosed in quotes. Atoms are scalar values of the language that have no internal structure. For example: `a foo '=' ':=' 'OZ 3.0' 'Hello World'`. Atoms have an ordering based on lexicographic ordering.

2.1.2 List

A *list* is either the atom `nil` representing the empty list, or is a tuple using the infix operator | and two arguments which are respectively the head and the tail of the list. Thus, a list of the first three positive integers is represented as: `1|2|3|nil`. A list ending by `nil` can also be represented by all elements between `[` and `]`, separated by a space and without the ending `nil`: `[1 2 3]` == `1|2|3|nil`.

2.1.3 Strings

Another notation for a list is a sequence of characters surrounded by `"` (double quotes), for example `"Hello World"`. This is equivalent to a list of integers where each integer is the ASCII value of the corresponding character in the *string*. This implies that all list operations are available for calculating with strings.

2.1.4 Records

Records are structured compound entities. A record has a label and a fixed number of components or arguments of the form `label(feat1:val1 ... featN:valN)` where `label` is an atom, the `featX` are atoms or numbers, and `valX` can be any valid data structure. Note that `featX` are optional. If not specified, they are implicitly numbered: `label (val1 ... valN) == label(1:val1 ... N:valN)`. A record whose features are numbered consecutively starting from 1 is called a *tuple*. Operations are provided to treat tuples in a special manner. Tuples give more concise code and have a faster implementation than records.

Many operations can be performed on Oz records (Table 1): let `R=toto(foo:10 bar:20)`.

Table 1. Some Oz operations.

Operation	Example
Selection	`R.foo == 10`
Get arity	`{Arity R} == [bar foo]`
Add feature	`{Record.adjoinAt R nuk 30} == toto(foo:10 bar:20 nuk:30)`
Subtract feature	`{Record.subtract R bar} == toto(foo:10)`
Extract label	`{Label R} == toto`
Rename label	`{Record.adjoin R lala} == lala(foo:10 bar:20)`

Iterations on record	`Record.forAll, Record.map, Record.while, ...`
	
	`{Record.map R fun{$ V} V div 10 end} == toto(foo:1 bar:2)`

Many other operations are available [5,8], and if they still do not cover the needs, the functional paradigm of Oz [17] can be used to write new ones compactly and efficiently. Because of dynamic typing, it is easy to create new record types at run-

time. As an example, we will describe two functions transforming the following record:

```
data(name:"Roger"
     surname:"Rabbit"
     address1:"Rue des toons"
     address2:"WB")
```

Function 1:

```
fun{Transform1 D}
   {List.toTuple td
    {List.map
     {Record.toListInd D}
     fun{$ I#E}
        lr(label(text:I)
           entry(init:E))
     end}}
end
```

The parameter `D` of the function is firstly transformed into a list of pairs: `featX#valX`. This list is mapped to a list where all elements have the form: `lr(label (text:featX) entry (init:valX))`, where `X` is the position of the item in the list. This list is transformed back into a tuple. The example record is thus transformed into (assuming implicit numbering):

```
td(lr(label(text:address1)
      entry(init:"Rue des toons"))
   lr(label(text:address2)
      entry(init:"WB"))
   lr(label(text:name)
      entry(init:"Roger"))
   lr(label(text:surname)
      entry(init:"Rabbit")))
```

Function 2:

```
fun{Transform2 D}
   fun{Loop P}
      case P of I#E|Xs then
         label(text:I)|
         entry(init:E)|
         newline|
         {Loop Xs}
      else nil end
   end
in
   {List.toTuple lr
      {Loop {Record.toListInd D}}}
end
```

Like `Transform1`, the function `Transform2` first transforms the record given as parameter into a list of pairs `featX#valX`. This list is then processed by the `Loop` function and the resulting list transformed back into a tuple whose label will be `lr`. The `Loop` function recursively parses a list of pairs and creates another list where

for one item `featX#valX` in the first list correspond three items in the second list: `label(text:featX)|entry(init:valX)|newline`. The resulting record is:

```
lr(label(text:address1)
   entry(init:"Rue des toons")
   newline
   label(text:address2)
   entry(init:"WB")
   newline
   label(text:name)
   entry(init:"Roger")
   newline
   label(text:surname)
   entry(init:"Rabbit")
   newline)
```

3 A Model-Based Approach Based on Oz Records

The idea of the model-based approach underlying QTk [7] is to map records describing starting models (typically, a domain model) onto widgets (typically, a presentation and a dialog model) according to the following mapping rules: a record is mapped onto a widget according to selection rules [19], its label is mapped onto the widget type, and any feature, e.g., to set the initial value, is mapped onto any widget parameter. The example `label(text:"surname")`declares a label widget which initially displays the text surname. The geometry management is defined by container widgets (`td` for top-down and `lr` for left-right) and not by a separate mechanism. Functions 1 and 2 of Section 2.1.4 give rise to the windows depicted in Fig. 1.

Fig. 1. Windows generated by applying mapping rules.

We briefly introduce the main mechanisms used by QTk to describe and integrate UI description records. These mechanisms make QTk a complete and useful toolkit for designing UIs in a model-based approach. [7] is a complete description of QTk.

3.1 Geometry Management

The geometry management is done by means of container widgets. The two most common container widgets are `td` and `lr` widgets which organize the contained

widgets respectively top to down and left to right. Fig. 2 shows the two windows that QTk builds with the following two records:

```
lr(label(text:"left")
   label(text:"center")
   label(text:"right"))
td(label(text:"top")
   label(text:"center")
   label(text:"down"))
```

Fig. 2. Windows generated from `lr` and `td` expressions.

Any of the contained widgets can recursively be containers. These two widgets can split a window into packed rectangular areas. To determine what size these areas must occupy, each widget has a `glue` parameter that place constraints on them. Without going into full details, one can choose that either a widget should occupy only the size required in a specific direction, or take as much space as possible. One can also choose to stick widgets to none, one, or more of its four possible edges to "`glue`" its neighbors or container border.

```
lr(label(text:"Name" glue:w) entry(glue:we) glue:nwe)
```

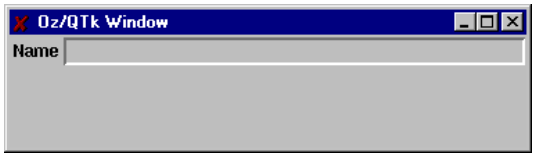

Fig. 3. Windows generated from a `glue` instruction.

The example reproduced in Fig. 3 shows a `lr` container widget glued so that it expands horizontally and is stuck to its container top edge. The label widget is glued so that its left edge is stuck to the border of the container. The entry expands horizontally taking all remaining available size left from the container.

3.2 Geometry Management: The Grid Structure

It is possible to have a grid structure where all widgets are organized in lines or columns of the same size (Fig. 4). The `lr` (resp. `td`) widget supports the `newline` special code which makes the following contained widgets jump to a new line (resp. column) right below the previous widgets, keeping the same column structure (resp. line) with the widgets above them. The `empty` special code leaves an empty space in a line (resp. column) and the `continue` special code spans a widget over several

columns (resp. lines) (Fig. 4).

```
lr(button(text:"One" glue:we) button(text:"Two" glue:we)
   button(text:"Three" glue:we) newline
   button(text:"Four" glue:we) button(text:"Five" glue:we)
   button(text:"Six" glue:we) newline
   button(text:"Seven" glue:we) button(text:"Height" glue:we)
   button(text:"Nine" glue:we) newline
   empty button(text:"Zero" glue:we) continue)
```

Fig. 4. Window generated from `newline` and `continue` instructions.

3.3 Creation of a window

The function `build` of the QTk module takes a record as input and if the record is a valid description, it builds a window corresponding to the description and returns an object controlling that window: `Window={QTk.build td(label(text:"Hello world"))}`. `{Window show}` exposes the previously created window.

3.4 Interaction with Oz

A QTk description is a static record that has to be interpreted. For the application to gain dynamic control over its UI, handles can be defined for all widgets:

```
entry(handle:E)
```

When the window is built, each defined handle is bound to an object controlling its corresponding widget.

```
Window={QTk.build td(entry(handle:E))}
{Window show}
{E set(text:"Type here")}
```

This is where QTk places the border between the declarative approach and a classical imperative, possibly object-oriented, approach: declarative approach for building the window in an initial state, imperative approach for dynamical interaction with window components.

3.5 Dynamic Windows

QTk provides a widget, called `placeholder`, whose content can change dynami-

cally during UI execution. A placeholder widget defines a rectangular area in the window that can contain any other widget at any time as long as the window exists. In the following example, the window alternatively contains a label and a push button.

```
placeholder(handle:P)
...
{P set(label(text:"Hello"))}
...
{P set(button(text:"World"))}
```

3.6 Events

Some widgets have an action parameter that can be defined to execute an Oz procedure (or a method invocation) when a widget specific action occurs. For example, the action parameter of a push button executes the procedure when the user presses this button. For arbitrary events, widgets can invoke the bind method from their handles.

3.7 Summary of the Model-Based Approach of QTk

QTk can map a single correctly formatted record to build a window:

1. Many different types of windows can be built with different type of geometry management, including grid structures.
2. All initial states of widgets can be defined.
3. All widgets can be controlled by the application using handles.
4. All type of events supported by widgets can be managed.
5. Part of a window can be defined and changed at any time during execution by means of the placeholder widget.

It is thus reasonable to use QTk as a support toolkit for a model-based approach to designing executable UIs and their applications.

4 Integration of QTk and Oz

This section describes interesting points of the current implementation of QTk in Oz.

4.1 Building a UI on-the-fly with a Model-Based Approach

Using Oz records for building interfaces reduces the implementation issues to manipulating records:

- Records are natural Oz data structures with a complete support to extensively manipulate them: the problem of building an interface (at least in its initial

state) is reduced to natural data structure manipulations. In practice this reduces very much the amount of work required for building a window: clear and compact notation and easy manipulation with extensive support from the language.

- Records are part of the application itself. They can be fully integrated with the application's data in a natural way without requiring a special communication layer. The dynamic part of the UI is managed by a classical imperative approach.
- Records can be built on-the-fly: an application can therefore completely create its UI at runtime from its model. This is a very important property: the model-based UI generation can be done both at design time and runtime, using one and only one programming language.
- Dynamic building of UI is a matter of creating a correctly formatted record on-the-fly. As Oz extensively supports this kind of manipulation, creating a UI at runtime is much simpler with Oz than with a classical imperative approach where developers manipulate many lines of code describing the dynamic UI creation rather than data structures.
- Records can be embedded inside Oz objects or procedures. It is consequently easy to write several widgets having different presentations for the same data structure, with a common interface at the Oz level. Fig. 1 shows two (slightly) different presentations of the same data structure. If various presentations are embedded inside objects or procedures, a selection rule can choose which presentation to use. Note that this selection is completely done at runtime. Writing such a selection rule in QTk is significantly shorter than with traditional imperative languages. It is estimated that a QTk selection rule would require one fourth of the code required in, for instance, SEGUIA [19]. This system contains about 360 selection rules written in imperative programming. Fig. 5 in Appendix shows a dialog box that can be dynamically adapted by change its interaction: either as read-only display or an input/output dialog box. These two presentations are embedded in two separate functions with their own handling and offer a common interface. Moreover, they are put in a placeholder so that the application can switch between the two states on-the-fly. The complete Oz code is given to demonstrate the compactness of the required code. Again, coding this UI with a traditional imperative programming language would require a significantly larger portion of code.

4.2 Porting QTk to Other Languages

Four language entities of Oz are used extensively by QTk: records (along with their manipulation operations), objects (for controlling widgets), procedures (for defining actions), and threads (for window concurrency).

1. The declarative specification uses records and is built using the record-manipulation operations in the language. The functional programming part of Oz is particularly useful for this. The declarative style is expressive enough to specify three basic properties:

- The widget types and their initial states.
- The initial actions, specified as procedure values (in the functional style).
- The geometric arrangement of widgets in the window and their behavior upon window resizing.

2. The active behavior of the UI is modeled using objects and threads. Each widget has a "handle", an object that can be used to control and interrogate the widget. Each window is associated with a thread. All actions of the window are executed sequentially in that thread. This guarantees that order of events within a window is not lost and that windows can operate autonomously. Threads in Oz are not operating system threads but are implemented by the Oz runtime system. They are designed to be extremely lightweight, thus allowing them to be used when the architecture of the application requires it, without having to be particularly concerned about efficiency.

Porting QTk to a more usual language like Java would require implementing two separate program components:

1. A package to support dynamically-typed records and allow their manipulation in a similar way to higher-order functional programming.
2. An interpreter, the analogue of QTk, that takes the records as input and interfaces with a low-level toolkit such as AWT. QTk and the low-level Tk interface together consist of about 13,000 lines of Oz. QTk fully uses the abilities of the Oz language, including its flexible object system (e.g., first-class messages), symbolic data structures (records and lists), and higher-order (functional) programming style where appropriate. It is more difficult to estimate the effort needed in duplicating the record functionality, since records are an integral part of the Oz system and their functionality is spread out. The record functionality consists of record input/output, an efficient internal record representation, a wide variety of primitive record manipulation operations, and hooks into the garbage collector.

However, this port would retain three disadvantages:

1. *Cumbersome syntax.* The record syntax provided by a Java package would almost certainly be more verbose than the Oz record syntax, unless an Oz-like parser is built into the package. Furthermore, since in-line procedures are not possible in Java, they would have to be defined separately and then referenced in the record.
2. *Inefficient records.* Oz records are carefully implemented to be efficient in both time and memory. They require one word per feature and label. If the record type and feature name are known at compile-time, then the field’s content is accessed by direct indexing (similar to statically-typed languages). If they are not, then the following techniques are used to get almost the same performance. For *tuples* the field is accessed by direct indexing, for records either by hashing (first access of a field) or by direct indexing (subsequent accesses of a field, with cached index [5]).
3. *Inefficient threads.* Building "plastic widgets" or other dynamic interface elements is straightforward in QTk because Oz threads are highly reactive and

lightweight.[1] Thread creation in Mozart 1.1.0 is two orders of magnitude faster than in JDK 1.2 (some timing figures are given in [7]). In Java, one would therefore have to build an explicit task scheduler: create the threads once and then give them tasks to do.

5 Related Work

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 UIs. We now compare QTk with some selected work regarding to the executability of models and regarding to other model-based approach.

Jacob [9,10] 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+ [15] 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 [2], 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 [16] 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 [13] 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 goes beyond the simple facilities provided by Tcl/Tk by offering the power of the

[1] Lightweight threads are important for constraint programming, which was one of the original target domains of the Oz language.

Oz language to UI designers and developers.

Bumbulis *et al.* 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 [3]. 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.

Miyashita *et al.* 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.

6 Conclusion

Due to facilities explained before and due to its implementation, QTk offers four original and substantive advantages:

1. *Unicity of language*: only one language serves as both modeling and programming language. This provides a significant advantage over most existing model-based UI development environment where the interface modeling language is significantly different from the target programming language. In QTk, the interface modeling language **is** the programming language and vice versa. Moreover, all mappings between potential models contained in one Oz program can be supported by the language itself. The example provided in the appendix shows how some reasonably complex UIs can be written with QTk.
2. *Reduced development cost*: the interface model immediately gives rise to an executable user interface. There is no longer a need for translating specifications contained in the models into code of the target programming environment.
3. *Tight integration of tools*: specification, construction, and execution tools are all integrated. There is no longer a need to distinguish the categories of tools that are often prevalent in model-based approaches. There is only one developing environment. Moreover, once the UI has been produced, the semantic core component can be developed without leaving the environment and without transforming the UI models into another expression. Rarely in the field of Human-Computer Interaction or in the domain of model-based approaches, such a tight integration between modeling and developing has been reached.
4. *Improved expressiveness*: the interface model is very compact to produce and cheap to manipulate. In most cases, the code needed to express the rules that gov-

ern the presentation and the dialogue of a running UI is significantly reduced with respect to fully imperative languages. For example, adaptation rules can be written in a very straightforward way that is more expressive than, for instance, its corresponding code in Java or C++. It is to a point that an adaptation rule can even be read, and thus interpreted, by just reading the Qtk code.

As QTk is built on top of Tcl/Tk [13], all its declarations are transformed into 100% imperative Tcl scripts. This is time consuming as the Oz process has to build a script, send it to the Tcl/Tk process, and Tcl/Tk has to interpret and process it. Moreover QTk has to interpret the UI declarations which is also time consuming, but that has to be done only once at the creation of the window (or when something new is put in a placeholder). The first overhead will disappear very soon as an equivalent to the QTk module is being currently developed for the GTk toolkit, where access to the toolkit is achieved by direct call to its API instead of an intermediate scripting language. The second overhead cannot be removed completely as there still will be an interpretation part when interacting through actions and handles, but we expect it to be small as this interaction is quite straightforward and does not require much computation.

Future work includes the development of mapping rules that support multiple forms of adaptation, including adaptativity, adaptability and plasticity. The example provided in the appendix is an example of such a UI with dynamic adaptation.

Acknowledgements

We thank all Mozart developers and contributors for their hard work in developing Oz and the Mozart platform. The development of QTk would not have been possible without their work. In particular, we thank Christian Schulte, the author of the Oz object-oriented binding to Tk that underlies QTk. This research is partly financed by the Walloon Region of Belgium.

References

1. Alexander, H.: Executable Specifications as an Aid to Dialogue Design. In: Proc. of IFIP International Conference on Human-Computer Interaction Interact'87 (1987) 739–744
2. 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, v. 2 (1996) 254–255
3. 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.): Proc. of Eurographics Workshop on Design, Specification and Verification of Interactive Systems DSV-IS'95 (Bonas, June 7-9, 1995). Springer-Verlag, Vienna (1995) 174–192
4. Cockton, G.: Using the Human Context in Interactive Systems Development. In: Unger,

C., Bass, L.: Proc. of Engineering for Human-Computer Interaction EHCI'95. Chapman and Hall (1995) 339–347
5. Duchier, D., Kornstaedt, L., Schulte, Ch.: The Oz Base Environment. (February 7, 2000). Accessible at http://www.mozart-oz.org/documentation/base/index.html
6. Gram, Ch., Cockton, G. (eds.): Design Principles for Interactive Software. Chapman & Hall, London (1996)
7. Grolaux, D.: QTk Module. (March 13, 2000). Accessible at http://www.info.ucl.ac.be/people/ned/qtk/http-html/index.html
8. Haridi, S., Franzén, N.: Tutorial of Oz. (February 7, 2000). Accessible at http://www.mozart-oz.org/documentation/tutorial/index.html
9. Jacob, R.J.K.: Executable Specifications for a Human-Computer Interface. In: Proc. of ACM Conference on Human Factors in Computing Systems CHI'83 (Boston, December 12-15, 1983). New York, ACM Press (1983) 28–34
10. 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) 211–242
11. 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) 107–116
12. Mozart Consortium: The Mozart Programming System (Oz 3). (January 1999). Accessible at http://www.mozart-oz.org
13. Ousterhout, J.K.: Tcl and the Tk Toolkit. Addison-Wesley, Reading (1994)
14. Paternò, F., Model-based Approach, Springer-Verlag, Berlin, 1999.
15. 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) 41–46
16. Puerta, A., Eisenstein, J.: Towards a General Computational Framework for Model-based Interface Development Systems. In: Proc. of ACM Conference on Intelligent User Interfaces IUI'99 (Los Angeles, January 1999). ACM Press, New York (1999), 171–178
17. Smolka, G.: The Oz Programming Model. In: Lecture Notes in Computer Science, vol. 1000. Springer-Verlag, Berlin (1995)
18. Szekely, P., Luo, P., Neches, R.: Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design. In: Proc. of ACM Conference on Human Factors in Computing Systems CHI'92 (Monterey, May 3-7, 1992). New York, ACM Press (1992) 507–515
19. Vanderdonckt, J., Bodart, F.: Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection. In: Proc. of the ACM Conference on Human Factors in Computing Systems InterCHI'93 (Amsterdam, April 24-29, 1993). ACM Press, New York (1993) 424–429

7 Appendix

The following example will show how two representations can be dynamically calculated on the same set of data. The user has the opportunity to freely switch between both views by clicking a check button (Fig. 5). For simplicity reasons and as the object-oriented notation of Oz wasn't introduced in this paper, this example uses a functional paradigm. The OO paradigm can be even more suited for this kind of UI development [12].

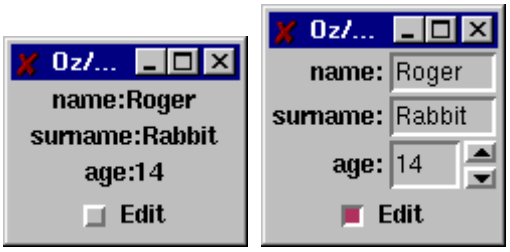

Fig. 5. Two representations with the opportunity to switch dynamically between them.

This application uses two functions to calculate two distinct views and encapsulate the access to the inner widget through a common interface. A placeholder is used to dynamically display one of these two views. The common interface makes switching and keeping the coherence between views trivial.

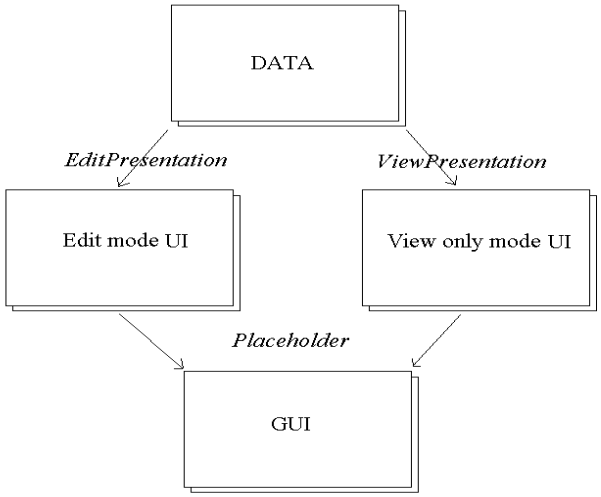

Fig. 6. From the initial data to the user interface.

7.1 Commented code

First of all, the application must declare it wants to use the QTk module. This module is available through an URL:

```
declare
[QTk]={Module.link
["http://www.info.ucl.ac.be/people/ned/qtk/QTk.ozf"]}
```

The data to work on is a list of pairs of `identifier#value`:

```
Data=[name#"Roger"
      surname#"Rabbit"
      age#14]
```

The `ViewPresentation` function builds a representation where each `pair identifier#value` is mapped to a label whose text is `identifier` followed by ":" and by the corresponding `value`. The functions returns a record with four fields:

- `desc`: the description record.
- `handle`: the handle of this representation.
- `set`: a one parameter procedure that refreshes the representation with the information that is given as parameter.
- `get` : a function that returns the state of the displayed data. Note that for simplicity, this state is the last one stored in the In variable, and not deduced from the displayed widgets.

```
fun{ViewPresentation Data}
   Handle In={NewCell Data} in
   r(desc:{Record.adjoin {List.toTuple td {List.map Data
        fun{$ D#V} label(glue:we feature:D text:D#":"#V) end}}
           td(glue:nswe handle:Handle)}
     handle:Handle
     set:proc{$ N} {Assign In N}
        {ForAll N proc{$ D#V} {Handle.D set(text:D#":"#V)} end}
         end
     get:fun{$} {Access In} end)
end
```

The `EditPresentation` function builds a representation where each `pair identifier#value` is mapped to a label containing `identifier` followed by ":" and an entry (if `value` is a string) or `numberentry` (if `value` is an integer) containing `value`. This representation uses a gridded structure, very much like example 2 in Fig. 1. This function returns a record with four similar fields to `the ViewPresentation` function. Note that this time, the data returned by the `get` function is deduced from the widgets themselves.

```
fun{EditPresentation Data}
   Handle Feats={List.map Data fun{$ D#_} D end}
   fun{Loop X} case X of D#V|Xs then
       label(glue:e text:D#":")|if {IsInt V}
       then numberentry(feature:D init:V glue:we)
       else entry(feature:D init:V glue:we)
```

```
      end|newline|{Loop Xs} else nil end
   end in
   r(desc:{Record.adjoin {List.toTuple lr {Loop Data}}
           lr(glue:nswe handle:Handle)}
     handle:Handle
     set:proc{$ N}
            {ForAll N proc{$ D#V} {Handle.D set(V)} end} end
     get:fun{$}
       {List.map Feats fun{$ D} D#{Handle.D get($)} end} end)
end
```

The main application calls both functions on the same data. The main window contains a placeholder and a check box. Checking or unchecking the box switches between both views, maintaining data integrity between views by using their associated `set` and `get` procedure and function. After the window is built, the descriptions of both views are placed in the window. Thereafter they can be placed back at any time by using their handles.

```
V1={ViewPresentation Data}
V2={EditPresentation Data}
P C
{{QTk.build td(placeholder(glue:nswe handle:P)
               checkbutton(text:"Edit" init:false handle:C
                           action:proc{$}
                                     Old#New=if {C get($)} then
V1#V2 else V2#V1 end
                                     V={Old.get} in
                                     {New.set V}
                                     {P set(New.handle)}
                                  end)} show}
{P set(V2.desc)} {P set(V1.desc)}
```

7.2 Uncommented code

```
declare
[QTk]={Module.link
["http://www.info.ucl.ac.be/people/ned/qtk/QTk.ozf"]}
Data=[name#"Roger"
      surname#"Rabbit"
      age#14]
fun{ViewPresentation Data}
   Handle In={NewCell Data} in
   r(desc:{Record.adjoin {List.toTuple td {List.map Data
        fun{$ D#V} label(glue:we feature:D text:D#":"#V) end}}
           td(glue:nswe handle:Handle)}
     handle:Handle
     set:proc{$ N} {Assign In N}
        {ForAll N proc{$ D#V} {Handle.D set(text:D#":"#V)} end}
         end
     get:fun{$} {Access In} end)
end
fun{EditPresentation Data}
   Handle Feats={List.map Data fun{$ D#_} D end}
   fun{Loop X} case X of D#V|Xs then
       label(glue:e text:D#":")|if {IsInt V}
       then numberentry(feature:D init:V glue:we)
       else entry(feature:D init:V glue:we)
       end|newline|{Loop Xs} else nil end
   end in
   r(desc:{Record.adjoin {List.toTuple lr {Loop Data}}
           lr(glue:nswe handle:Handle)}
     handle:Handle
     set:proc{$ N}
           {ForAll N proc{$ D#V} {Handle.D set(V)} end} end
     get:fun{$}
       {List.map Feats fun{$ D} D#{Handle.D get($)} end} end)
end
V1={ViewPresentation Data}
V2={EditPresentation Data}
P C
{{QTk.build td(placeholder(glue:nswe handle:P)
             checkbutton(text:"Edit" init:false handle:C
                      action:proc{$}
                               Old#New=if {C get($)} then
V1#V2 else V2#V1 end
                               V={Old.get} in
                               {New.set V}
                               {P set(New.handle)}
                            end)} show}
{P set(V2.desc)} {P set(V1.desc)}
```