Embedding a Scheme Interpreter in the Tk Toolkit

Erick Gallesio
Université de Nice - Sophia-Antipolis
Laboratoire I3S - CNRS URA 1376 - Bât 4.
250, avenue Albert Einstein
Sophia Antipolis
06560 Valbonne - FRANCE

Abstract

STk *is a graphical package which rely on Tk and the Scheme programming language. Concretely, it can be seen as the Tk package where the Tcl language as been* replaced *by a Scheme interpreter. Programming with* STk *can be done at two distinct levels. First level is quite identical than programming Tk with Tcl. Second level of programming uses a full object oriented system. Those two programming levels and current implementation are described here.*

1 Introduction

Today's available graphical toolkits for applicative languages are not satisfactory. Most of the time, they ask to the user to be an X expert which must cope with complicated arcane details such as server connections or queue events. This is a true problem, since people which use this kind of languages are generally not inclined in system programming and little of them get over the gap between the language and the toolkit abstraction levels.

Tk is a powerful X11 graphical toolkit defined at the University of Berkeley by J.Ousterhout [?]. This toolkit gives to the user high level widgets such as buttons or menu and is easily programmable. In particular, a little knowledge of X fundamentals are needed to build an application with it. Tk package rely on an interpretative language named Tcl [?]. However, dependencies between those two packages are not too intricate and replacing Tcl by an applicative language was an exciting challenge. To keep intact the Tk/Tcl pair spirit, a little applicative language was necessary. Scheme [?] was a good candidate to replace Tcl, because it is small, clean and well defined since it is an IEEE standard [?].

Programming with STk can be done at two distinct levels. First level is quite identical than programming Tk with Tcl, excepting several minor syntactic differences. Second level of programming uses a full object oriented system (with multi-inheritance, generic functions and a true meta object protocol). Those two levels of programming are briefly described in the two first sections. Section 4 is devoted to implementation and section 5 exposes some encountered problems when mixing Tk and Scheme.

2 STk : First level

The first level of STk uses the standard Scheme constructs. To work at this level, a user must know a little set of rewriting rules from the original Tk-Tcl library. With these rules, the Tk manual pages and a little knowledge of Scheme, he/she can easily build a STk program.

Creation of a new widget (button, label, canvas, ...) is done with special STk primitives procedures. For instance, creating a new button can be done with

```
(button ".b")
```

Note that the name of the widget must be "stringified" due to the Scheme evaluation mechanism. The call of a widget creation primitive defines a new Scheme object. This object, which is considered as a new basic type by the Scheme interpreter, is automatically stored in a variable whose name is equal to the string passed to the creation function. So, the preceding button creation would define an object stored in the `.b` variable. This object is a special kind of procedure which is generally used, as in pure Tk, to customize its associated widget. For instance, the expression

```
(.b 'configure
    '-text "Hello, world"
    '-border 3)
```

permits to set the text and background options of the `.b` button. As we can see on this example, parameters must be well quoted in regard of the Scheme evaluation rules. Since this notation is barely crude, the Common Lisp keyword mechanism has been introduced in the Scheme interpreter [?] [1]. Consequently, the preceding expression could have been written as

```
(.b 'configure
    :text "Hello, world"
    :border 3)
```

which in turn is both close from Common Lisp and pure Tk. Of course, as in Tk, parameters can be passed at widget creation time and our button creation and initialization could have been done in a single expression:

```
(button .b
        :text "Hello, world"
        :border 3)
```

The Tk binding mechanism, which serves to create widget event handlers follow the same kind of rules. The body of a Tk handler must be written, of course, in Scheme. Following example shows such a script; in this example, the label indicates how many times mouse button 3 has been depressed. Button press counter increment is achieved with the simple script given in the `bind` call.

```
(define count 0)
(pack 'append "."
      (label ".l"
             :textvariable 'count)
             "fill")
(bind .l "<3>"
         '(set! count (+ count 1)))
```

[1] A keyword is a symbol beginning with a colon. It can been seen as a symbolic constant (i.e. its value is itself)

To illustrate STk first level of programming, Figure 1 shows the simple file browser described in [?] written in STk .

3 STk : Second level

Programming with material shown before is a little bit tedious and more complicated than coding with Tcl since one have to add parenthesis pairs and quote options values. Its only interest is to bring the power and flexibility of Tk to the Scheme world.

The second level of STk is far more interesting since it uses a full object oriented extension of the Scheme language. Defining an object oriented layer on Scheme is a current activity in the Scheme community and several packages are available. The object layer of STk is derived from a package called *Tiny Clos* [?]. This extension provides objects *à la* CLOS (Common Lisp Object System). In fact, the proposed extension is much closer from the objects one can find in Dylan, since this language is already a tentative to merge CLOS notions in a Scheme like language [?].

STk object extension gives to the user a full object oriented system with multi-inheritance and generic functions. Furthermore, all the implementation rely on a true meta object protocol, in the spirit of [?]. This model has been used to embody all the predefined Tk widgets in a hierarchy of Stk classes.

3.1 Class hierarchy

With the STk object system, every Tk graphical object used in a program such as a menu, a label or a button is represented as an object in the Scheme core. All the defined STk classes build a hierarchy which is briefly described here. Firstly, all the classes shared a unique ancestor: the *<Tk-object>* class. Instances of this class contain informations which are necessary to establish a communication between the Scheme and Tk worlds. Objects of this class have two main slots named `Id` and `parent`. The `Id` slot contains a string, normally generated by the system, which correspond to a (unique) variable name in

```
#!/usr/local/bin/stk -f

(scrollbar ".scroll" :command ".list 'yview")
(listbox ".list" :yscroll ".scroll 'set" :relief 'raised :geometry "20x20")
(pack 'append "." .scroll "right filly" .list "left expand fill")

(define (browse dir file)
  (if (not (string=? dir ".")) (set! file (string-append dir "/" file)))
  (if (directory? file)
      (system (format #f "browse.stk ~A &" file))
      (if (file? file)
          (let ((ed (getenv "EDITOR")))
            (if ed
                (system (string-append ed " " file "&"))
                (system (string-append "xedit " file "&"))))
          (error "Bad directory or file" file))))

(define dir (if (> argc 0) (car argv) "."))

(system (format #f "ls -a ~A > /tmp/browse" dir))

(with-input-from-file "/tmp/browse" (lambda()
  (do ((f (read-line) (read-line)))
      ((eof-object? f))
    (.list 'insert 'end f))))

(bind .list "<Control-c>" '(destroy "."))
(bind .list "<Double-Button-1>" '(browse dir (selection 'get)))
```

Figure 1: *A simple file browser*

the Tk world. The `parent` slot contains a reference to the object which (graphically) includes the current object. Normally, end users will not have to use direct instances of the <*Tk-object*> class[2].

The next level in our class hierarchy defines a fork with two branches: the <*Tk-widget*> class and <*Tk-canvas-item*> class. Instances of the former class are classical widgets such as buttons or menus since instances of the later are objects contained in a canvas such as lines or rectangles. Tk widgets are also divided in two categories: <*Tk-simple-widgets*> and <*Tk-composite-widgets*>. Simple widgets are directly implemented as Tk objects and composite ones are built upon simple widgets (e.g. file browser, alert messages and so on). A partial view of the STk hierarchy is shown in Figure 2.

[2] All classes whose name begins with the "Tk-" prefix are not intended for the final user.

3.2 Basic notions

This section describes basic concepts of our object extension on a small example. Definition of a new object class is done with the *defclass* form. For instance,

```
(defclass person ()
   ((name :initarg :name
          :accessor name
    (age  :initarg :age))))
```

defines a person characteristics. Two slots are declared: `name` and `age`. Characteristics of a slot are expressed with its definition. Here, for instance, it is said that the slot `name` can be inited with the keyword `:name` upon instance creation and that an accessor function should be generated for it. Creation of a new instance is done with the `make` constructor:

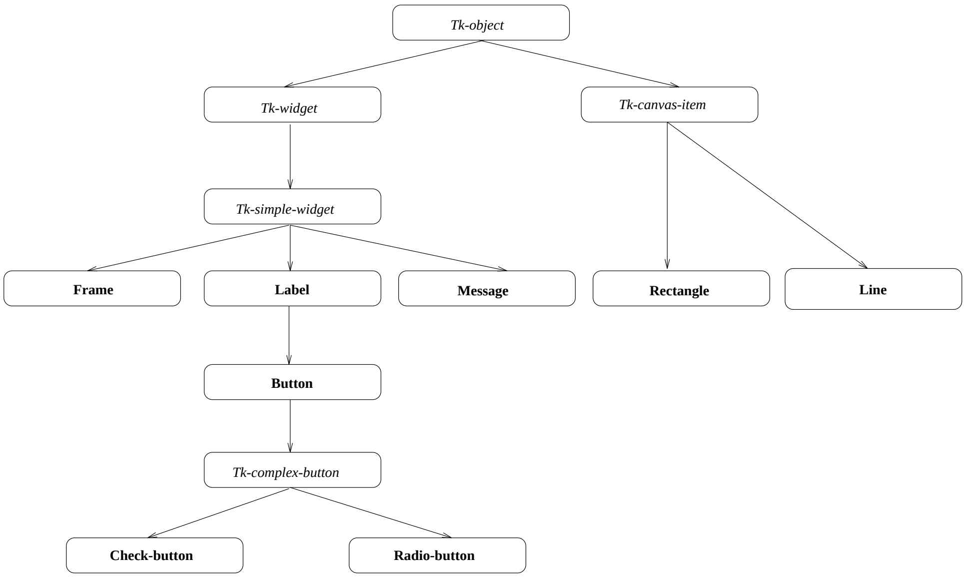

Figure 2: *A partial view of* STk *hierarchy*

```
(define p (make person
                :name "Smith"
                :age 42))
```

This call permits to build a new person and to initialize the slots which compose him/her.

Reading the value of a slot can be done with the function `slot-value`. For instance,

```
(slot-value p 'age)
```

permits to get the value of slot `age` of the `p` object. Setting this slot can be done by using the generalized `set!` defined in STk :

```
(set! (slot-value p 'age) 43)
```

Since an accessor as also been defined on the `name` slot, it can be read with the following expression:

```
(name p)
```

As before, slot setting can be done with the generalized `set!` as in

```
(set! (name p) 43)
```

3.3 Tk classes

Now that basic concepts have been exposed, let come back to how using Tk with the object layer. In our model, each Tk option is defined as a slot. For instance, a simplified definition of a Tk button could be:

```
(defclass <Button> (<Label>)
   ((command :accessor command
             :initarg :command
             :allocation :pseudo
             :type 'any))
   :metaclass <Tk>)
```

This definition says that a <Button> is a (*inherits* from) <Label> with an extra slot called `command`. This slot's allocation scheme is said to be `:pseudo`[3]. Pseudo-slots are special purpose slots: they can be used as normal slots but they are not allocated in the Scheme world (i.e. their value is stored in one of the structures manipulated by the Tk library instead of in a Scheme object). Of course, accessors will take into account this fact and functions for reading or writing such slots are unchanged. For example,

```
(set! (command b) '(display "OK"))
```

permits to set the script associated to the `b` button.

Preceding `defclass` states that the `command` slot can contain a value of any type. Type of a slot permits to the system to apply the adequate coercion function when a slot is read. Since Tk

[3]Pseudo-slots are defined in the metaclass <Tk>, hence the `:metaclass` option in definition.

always computes results as strings, this conversion can be done automatically when we know the type of the slot (e.g. a border width is always stored as an integer in Tk structures). No conversion is done when a slot is written; this work is done by Tk since it will reject bad values.

Note that using the object extension of STk permits the user to forget some Tk idiosyncrasies. In particular, it permits to avoid the knowledge of pure Tk naming conventions. The only thing the user has to know when creating a new object is it's parent. An example of widgets creation is shown below:

```
(define f (make <Frame>))
(define b1 (make <Button>
                 :text "B1"
                 :parent f))
(define b2 (make <Button>
                 :text "B2"
                 :parent f))
```

Created buttons here specify that their parent is the frame `f`. Since this frame does not specify a particular parent, it is supposed to be a direct descendant of the root window `"."`. This parent's notion is also used for canvas items: a canvas item is considered as a son of the canvas which contains it. For instance,

```
(define c (make <Canvas>))
(define r (make <Rectangle>
                :parent c
                :coords '(0 0 50 50)))
```

defines a rectangle called `r` in the `c` canvas. User can now forget that `r` is included in `c` since this information is embedded in the Scheme object. To move this rectangle, one can use for example the following expression:

```
(move r 10 10)
```

which is more natural than the things we have to do at STk first level.

3.4 Generic functions

With the STk object layer, execution of a method doesn't use the classical message sending mechanism as in numerous object languages but generic functions. The mechanism implemented in STk is a subset of the generic functions of CLOS. As in CLOS, a generic function can have several methods associated with it. These methods describe the generic function behaviour according to the type of its parameters. A method for a generic function is defined with the *define-method* form.

Following example shows three methods of the generic function `value-of`:

```
(define-method value-of ((obj <Scale>))
  (string->number ((Id obj) 'get)))

(define-method value-of ((obj <Entry>))
  ((Id obj) 'get))

(define-method value-of (obj)
  (error "Bad call: " obj))
```

When calling the `value-of` generic function, system will choose the more adequate method, according to its parameter type: if parameter is a scale or an entry, first or second method is called; otherwise the third method is called since it doesn't discriminates in favour of a particular parameter type.

Setter method are a special kind of methods which are used with the generalized `set!`. Here are the corresponding setter methods to previous `value-of`:

```
(define-method (setter value-of)
               ((obj <Scale>) value)
  ((Id obj) 'set value))

(define-method (setter value-of)
               ((obj <Entry>) value)
  ((Id obj) 'delete 0 'end)
  ((Id obj) 'insert 0 value))

(define-method (setter value-of) (obj)
  (error "Bad call: " obj))
```

One of these methods will be called when evaluating following form, depending of type of `x`:

```
(set! (value-of x) 100)
```

As we can see here, generic functions yield things more homogeneous than what we can have at STk first level. Indeed getting and setting the value of an entry or a scale can now be done in a similar fashion with those methods.

4 Implementation - Performances

The STK interpreter is written in C and for some parts in Scheme. The object oriented layer presented in section 3 is totally written in Scheme. The Scheme interpreter is as far as possible conform with R4RS [?]. It is important to note that the Tk library is used unmodified in this interpreter. All the Tcl functions call issued by Tk primitives are simulated by STK . This permits to the STK interpreter to be, as far as possible, independent of the Tk code. In particular, embedding a new release of Tk will only require to link the new library to the actual STK interpreter. Furthermore, external contributions we can find on net can be easily included in the interpreter, since Tcl "intrinsics" replacements are present in the STK core.

Actual implementation doesn't put accent on performances. It must be seen as a prototype which must be stretched further. However, measuring the performances of the STK package is a difficult task and has not been really done yet. What we can say for now is that there is a little overhead when calling a Tk primitive written in C since the STK package must translate all the parameters in C strings. This translation must be done because the Tk library "thinks" that it works on Tcl which uses strings for passing parameters. In counterpart, procedure written in Scheme are far more efficient than Tcl scripts since the Scheme interpreter uses an appropriate format which is cheaper than strings. In particular, there is no data conversion when other Scheme procedures are called. Using the object extension of STK gives this tool more power but is, as we can expect, more time consuming. For now, penalty when using the STK object oriented layer is mainly due to object creation: creating a widget with a `make` is nearly 20 times slower than basic creation achieved by using only first level primitives. However, getting or reading a Tk option using a slot access is only 1.5 times slower than direct Tk configuration. We can expect that rewriting some parts of the STK object layer in C will decrease those ratios. Comparing the Tcl and Scheme approaches needs much more working; and a complete study will be done when the package will be more stable.

5 Open problems

Using Scheme with Tk causes some difficulties which cannot satisfactorily be resolved. Some problems are due to the very nature of Scheme, others are due to Tk. Following, is a list of major of them

- R4RS requires that symbol must be case insensitive. Tk imposes, in a certain extent, to the underlying interpretative language to be case sensitive since command in event handler scripts take into account the case of the letter following the % symbol.
- Another problem arises with lists and strings: Tcl doesn't distinguish those two types. In particular, one can set an option as a string and asking to Tk this option value will yield a list. This is not a problem in Tcl since a list is only another vision of a string. Unfortunately, there is no such direct equivalence in Scheme. Improvement of this point would probably require a modification of the Tk library.
- One of the major assets of the Tk library is the `send` command. With this command, a Tk application can ask to another running Tk interpreter to evaluate an expression. Since Tk library is not modified, STK interpreters cannot be distinguished from Tcl ones. Most of the time, this is confusing and error prone because requests to an application must take into account the language its underlying interpreter understands. Keeping the communication possibility between STK an Tcl applications seems to be suitable, but it would be fine to establish a standard way to determine what kind of interpreter is running in a particular application.