Specifying the Behavior of Graphical Objects Using Esterel

Dominique Clément and Janet Incerpi
INRIA – Sophia Antipolis
06565 Valbonne Cedex, FRANCE

Specifying the behavior of graphical objects, such as menus, scrollbars, etc. is not an easy task. This is because one must deal with multiple input devices such as the mouse and keyboard. This makes the specification of such objects difficult to write and hard to maintain. We consider these objects as reactive systems that receive inputs and generate output after updating their internal state. We present here how one can use the Esterel language to write efficient, clean, and modular specifications of such systems. Esterel also provides for the reuseability of such specifications.

1. Introduction

The use of graphical user interfaces has led to much research on various aspects of them. Many systems address how one goes about building user interfaces. Typically these provide graphical objects (such as buttons, menus, etc.) that the user can combine and interface with his underlying application. Specifying the behavior of these primitive graphical objects is not easy. This is because one quickly falls to the level of worrying about how to deal with mouse and keyboard input and various other interaction devices. While such events drive the comportment of graphical objects, the behavior should be expressed at a higher level without concern for how the underlying hardware (or low-level software) interfaces with such events.

We view graphical objects as reactive systems that respond to input events and generate output events. Here we present how one can specify cleanly and efficiently the behavior of graphical objects by using the Esterel [2] programming language. Esterel is a synchronous language for programming reactive systems; it combines the features of parallel languages with the efficiency of automata. Thus far we have only specified the behavior of graphical objects: buttons (trill, trigger), menus (pulldowns, pop-ups), sliders, scrollbars, menubars, etc. We would like to continue specifying higher-level and specialized objects. The results are promising. The specification is modular and easy to write and maintain. The resulting automata are very efficient. We separate the behavior from the graphics thus we obtain a hierarchy of behaviors independent of an underlying window system.

We begin with a description of the problem. A short presentation of Esterel follows; this can be skipped by those already familiar with Esterel. Next we present a small example of a button for discussing behavior of graphical objects and the kinds of events that are typical for such objects. In section 5, we present more complex examples highlighting the various aspects in specifying behavior. We then explain how one interfaces the generated automata with his system. Next we discuss related work. Finally we close with a discussion of future work.

2. The Problem

Graphical objects are the building blocks of user interfaces. These include buttons, menus, scrollbars, menubars, browsers, etc. There are many systems available for using and combining these objects ([10],[14],[12]) to build various interface components. This may include specifying layout when combining existing objects or specifying the appearance of a new object but for specifying the behavior the user is on his own. We feel that specifying the behavior of graphical objects is a complex task. It may require handling multiple input devices and often the notion of time is important. Furthermore, the problem of specifying a behavior must be attacked independently of the graphical objects themselves. That is, independent of the method used to define new objects.

In the simplest case, that of a button, we want that the description of the behavior is not dependent on the graphical aspects of the button and that this behavior can be reused. We are not concerned with the form of the button as it appears on the screen or its internal representation. For the behavior of a button one is concerned with relative position and, in the case of a "trill" button, with timing.

For more complex objects, it is typically not just a straightforward combination of simpler behaviors. Graphically a menu is a collection of buttons —typically appearing as a row or column. The behavior of a menu is *not* simply that of, say, a row of buttons. For a row of buttons one depresses a mouse button while inside a button then with the mouse one highlights and unhighlights only the chosen button deciding whether to perform the action by releasing the mouse button within the button. For a menu, while one initially depresses a mouse button in one of the menu's buttons to get things started, when moving to another button it is not necessary to click inside. Thus it is not sufficient to specify the behavior of a button. Clearly, we would like to reuse the button behavior adding the necessary control.

We feel that the problem of specifying the behavior must be attacked with an appropriate level of abstraction. For example, in the case of a menu one could ask what kind of menu do we want to specify? A fixed menu that is always on the screen? A pulldown menu or perhaps a pop-up menu? It is clear that once a pulldown or pop-up menu reveals its selections (buttons) that it behaves the same as a fixed menu. Thus, the difference is how one initiates the behavior. We want to write the specification for a menu which works for all three types of menus.

With even more complex objects, such as menubars or scrollbars, the behavior becomes increasingly more difficult to specify correctly. Typically the situation becomes so complex that one just decides to simplify the behavior. For example, in scrolling if one leaves the shaft of the scrollbar and re-enters then the thumb (or index) does not begin moving once again towards the mouse.

Efficiency is always a concern for interface components. One could directly hand-code the behavior as automata but we feel this is a difficult task even for simple behaviors. We find that with Esterel the generated automata are small and efficient.

3. Esterel

Esterel is a synchronous language designed to program reactive systems; it combines the features of parallel languages with the execution efficiency of automata. One can view an Esterel program as a collection of parallel processes which communicate instantly via broadcast signals. The underlying synchrony

hypothesis is that an Esterel program reacts instantly to its input by updating its state and generating output. It is the input and output that determine the behavior of the program.

Broadcast signals are the method by which Esterel communicates: internally using local signals and with its surroundings using input and output signals. There are two kinds of signals. With *pure* signals it is their presence (or absence) which is important. For example, one can emit a signal, say, "mouse-up" when a mouse button which was depressed is released. With *valued* signals there is typically some additional information which is important. In this case, one can emit a signal, say, "mouse" representing the mouse's position whose value would be the mouse coordinates. Signals in Esterel are simply identified by names. If S is a valued signal then $?S$ is its value. As well in Esterel there exists *sensors* which are valued inputs to a program that are solely queried and never emitted. A typical example of a sensor is the temperature, an Esterel program could query to find out what's the current temperature.

There is no notion of absolute time in Esterel. One can treat physical time as a standard signal. But more importantly one can treat every signal as a "time unit". One can introduce time in the appropriate form for a particular problem.

In Esterel programming, the *module* is the standard unit. Data is handled by abstract type facilities. The user declares types, signals, functions, and procedures. Valued signals have types, for example, a signal named *mouse* may be of type *point* or *coordinate*. Statements are of two types: those that are more classical dealing with assignments, functions, etc., and those that deal with signals.

Here we give a brief sketch of the types of constructs that are available. This is to aid the reader for the code that will be presented in later sections. Basic statements in Esterel include: assignment, if-then-else, loop-end (an infinite loop), procedure calls, and sequences or parallels of statements. The parallel construct requires that there are no shared variables. It is assumed (due to the synchrony hypothesis) that all these statements take no time. Also Esterel provides powerful exception handling mechanisms, in what follows we use a simple *trap-exit* construct. The "trap" declares an exception while an "exit" raises an exception. In the simplest case, the body of the *trap-exit* construct is executed normally until a corresponding "exit" is encountered:

```
trap FINISHED in
    < statements >
end
```

Here FINISHED is the name of the exception. If within the statements one encounters an "exit FINISHED" then the trap terminates and execution continues at the following instruction.

Another widely used construct is the *copymodule* statement. The *copymodule* instruction provides for in-place expansion, possibly with signal renaming, of other modules. This allows one to reuse existing modules.

There is also a class of statements which are temporal† and involve signal handling. This group includes:

- *emit*, to broadcast a signal, and *await*, to listen for a signal.
- a *present* statement which checks for a signal's presence activating either a "then" part or an "else" part.

† These can be used to handle time and synchronization.

- a *do-upto* statement which executes its body until a signal is received at which point it is aborted.
- two loop constructs: a *loop-each* and a *every-do-end*. These both execute the loop body's statements and restart at the top of the loop each time a signal is received. (Note the difference is the first entry into the loop. The "loop-each" body is entered before the first signal is received, while the "every-do" waits until the first signal is received.)

Esterel promotes a programming style that is very modular. Modules emit signals and at the time are unaware who, if anyone, is listening. Thus for a given task one can write many little modules each of which performs some task. This along with renaming of signals can lead to a collection of reuseable Esterel modules. Note that Esterel also encourages transmitting as much information as possible as signals. In many cases one stays away from the variables and if-then-else style of programming. This is because signals can store values and the present-then-else construct exists. The main advantage is that while an *if* statement always results in a runtime test, a *present* statement is compiled more efficiently. For more details regarding the Esterel language the reader should see [3].

4. A Small Example: A Trigger Button

We present a small example of how to use Esterel for describing a trigger button, where the associated action is triggered on mouse-up, that is, when the mouse button is released. This example† gives a feel for the style of programming.

4.1. *The Button Module*

A trigger button behaves as follows: depressing a mouse button inside the button highlights it; moving out (resp. in) the button keeping the mouse button depressed causes the button to be unhighlighted (resp. highlighted); finally releasing the mouse button inside the button performs some action and unhighlights the button; if the mouse button is released outside of the button then we are done but there is nothing to do. Here we look at how to specify this behavior in Esterel.

For every Esterel module it is necessary to have some input and output signals. This permits the module to communicate to the external world. These signals are defined in the external interface. This includes the declaration of user defined types, external functions and procedures.

Note that for the button behavior we need to know the button, the mouse position, when the mouse button is released, and if the action is to be performed. Thus, we have the following declarations:

† A more detailed presentation of all the examples presented in this paper appears in [9].

```
module BUTTON :
type COORD,
     RECTANGLE,
     BUTTON;
function GET_RECTANGLE (BUTTON) : RECTANGLE;
procedure XOR () (BUTTON);
input BUTTON (BUTTON),
      MOUSE (COORD),
      MOUSE_UP;
relation BUTTON # MOUSE_UP,
         BUTTON => MOUSE,
         MOUSE_UP => MOUSE;
output PERFORM_ACTION (BUTTON);
```

Above we have the declarations for the user abstract data types: COORD (coordinate or point), RECTANGLE, and BUTTON that will be used in this module. The button module has three input signals: BUTTON whose value is of type BUTTON†, MOUSE whose value is of type COORD, and MOUSE_UP (a pure signal). It has but one output signal: PERFORM_ACTION whose value is of type BUTTON.

We assume the button, through GET_RECTANGLE an external function, can supply a rectangle which is its sensitive area. Note that this rectangle can be smaller or larger than the visual appearance of the button. Also there is one external procedure XOR‡ that takes as argument the button to be highlighted.

The *relation* section gives information about the relationships among the input signals. The first states that the BUTTON and MOUSE_UP signals are incompatible and never appear in the same instant. Thus one cannot begin and end the button module in the same instant. The other two are causality relations. The former states that whenever the BUTTON signal is present then a MOUSE signal will also be present. Similarly the latter states whenever MOUSE_UP is present then MOUSE is also present (it tells where MOUSE_UP has occurred). These relations represent assumptions about how signals will be received by the module.

The behavior of the trigger button can be seen as three tasks running in parallel.

```
   < tell whether inside or out of the button >
||
   < control highlighting of button >
||
   < watch for mouse-up then do what's necessary >
```

---

† The signal declarations give the signal name followed by the type of the signal in parentheses. We name both the signal and the type BUTTON. This is not a problem as there are separate name spaces for the types, signals, and variables.

‡ Procedures in Esterel have two argument lists: the first is for call by reference arguments the second for call by value. We should note here that in function and procedure declarations it is the *type* of arguments (and *type* of the returned value for functions) that are given.

One could view the behavior for a trigger button as simply the last two tasks, highlighting and handling mouse-up, running in parallel, however, both of these are dependent on knowing whether one is in or out of the button. Thus it seems natural to separate such a task and consider it as a third component needed in specifying the behavior of a button.

We assume there exists a module whose task is to say whether or not a point is in or out of a given rectangle. This module, check-rectangle, generates (output) signals IN and OUT each time the situation changes; one goes from being outside the rectangle to being inside and vice versa. To do this, check-rectangle needs two input signals: CHECK_RECTANGLE whose value is of type RECTANGLE and MOUSE whose value is of type COORD. Also check-rectangle can answer queries about the current situation. That is, whether we are in or out of the rectangle. For this we need an input signal AM_I_IN and two output signals YES and NO.

We begin by showing how to run the check-rectangle module in parallel with the components for controlling the highlighting and handling mouse-up. The component to < tell whether inside or out of the button > can be written in Esterel as follows:

```
emit CHECK_RECTANGLE (GET_RECTANGLE (? BUTTON));
copymodule CHECK_RECTANGLE [signal BUTTON_IN / IN,
                                   BUTTON_OUT / OUT]
```

The Esterel *copymodule* construct allows one to use other Esterel modules. This corresponds to an in-place expansion of the check-rectangle module possibly with signal renamings. Here we rename the IN (resp. OUT) signal to be BUTTON_IN (resp. BUTTON_OUT). Thus this instance of the check-rectangle module emits BUTTON_IN and BUTTON_OUT. Before starting the module we send a CHECK_RECTANGLE signal whose value, given by GET_RECTANGLE† an external function, is the sensitive area for the button. Note that check-rectangle uses the MOUSE signal to determine when one moves in and out of the given rectangle. (Broadcasting signals allows the Esterel code to be modular; one doesn't have to know who is listening and anyone who is listening can act accordingly.)

Consider how one specifies in Esterel the control for highlighting. Recall that initially the mouse is inside the button (a natural assumption if one clicks in the button to start). The control is an infinite loop: highlight, wait until the mouse is out of the button, unhighlight, wait until the mouse is inside the button. In Esterel we have the following:

```
loop
   call XOR () (? BUTTON);
   await BUTTON_OUT;
   call XOR () (? BUTTON);
   await BUTTON_IN
end
```

The first call to XOR highlights the button while the second unhighlights it. The check-rectangle module which is running in parallel emits the signals BUTTON_OUT and BUTTON_IN.

† Note that "? BUTTON" is the value of the input signal BUTTON; since this signal is of type BUTTON, it is a valid argument to GET_RECTANGLE.

For handling mouse-up one equally needs to know whether the mouse is inside the button or not. Here we wait for mouse up then query the check-rectangle module and after possibly performing some action the button module is terminated.

```
await MOUSE_UP do
    emit AM_I_IN;
    present YES then
        call XOR () (? BUTTON);
        emit PERFORM_ACTION (? BUTTON)
    end;
    exit THE_END
end
```

When MOUSE_UP arrives we simply emit AM_I_IN then see if YES, which would be emitted by the check-rectangle module, is present. If so, we unhighlight the button and emit PERFORM_ACTION. The *exit* is part of the trap-exit mechanism of Esterel, thus with a corresponding *trap* statement surrounding the three parallel tasks one exits completely the button module.

5. Reusability and More Examples

We present various examples that show how one can re-use the Esterel modules to make a hierarchy of behaviors. We begin with a description of a menu, followed by that of pulldown and popup menus, and finally a menubar.

5.1. *A General Menu Module*

We now want to specify the behavior for a menu. We describe a module, called menu-body, which doesn't know whether one is initially in or out of the menu body (buttons). Then we show how this module can be used to attain the various kinds of menus.

The external interface for the menu-body module introduces a new type, MENU, and a new input signal, MENU, of this type. As well we have two external functions: GET_MENU_RECTANGLE takes a menu as argument and returns the rectangle associated with the menu body, and GET_MENU_BUTTON takes a menu and a point and returns the menu button which the point is in. The Esterel declarations are as follows:

```
module MENU :
type COORD,
     RECTANGLE,
     BUTTON,
     MENU;
function GET_MENU_RECTANGLE (MENU) : RECTANGLE,
         GET_MENU_BUTTON (MENU, COORD) : BUTTON;
procedure XOR () (BUTTON);
input MENU (MENU),
      MOUSE (COORD),
      MOUSE_UP;
output PERFORM_ACTION (BUTTON);
relation MENU # MOUSE_UP,
         MENU => MOUSE,
         MOUSE_UP => MOUSE;
```

The behavior of the menu can be described as follows: when the mouse is inside the menu body (i.e., inside one of its buttons) then the selected button is highlighted. The selected button changes as the mouse moves within the menu body; when the mouse is outside the menu then no button is highlighted; on mouse-up, if inside the menu the currently selected button's action is performed. We specifiy this behavior with three tasks in parallel:

```
   < tell whether inside or out of the menu body >
||
   < keep track of current button >
||
   < wait for mouse-up then do what's necessary >
```

The first task is just an instance of the check-rectangle module:

```
emit CHECK_RECTANGLE (GET_MENU_RECTANGLE (? MENU));
copymodule CHECK_RECTANGLE [signal MENU_IN / IN,
                                   MENU_OUT / OUT]
```

The task of keeping track of the current button needs to know whether the mouse is inside the menu body or not. Thus it uses MENU_IN and MENU_OUT, once inside the menu we must maintain the active button. This can be expressed in Esterel as follows:

```
loop
   await immediate MENU_IN;
   var ACTIVE_BUTTON : BUTTON in
      do
         < Maintain and run active button >
      upto MENU_OUT;
      call XOR () (ACTIVE_BUTTON)
   end
end
```

The "await immediate" allows one to start correctly the maintaining of the active button when one is initially inside the menu body. The call to XOR is necessary because the active button which will be running within the *do-upto* statement is instantly aborted on hearing MENU_OUT thus the active button is still highlighted; only the menu knows this and can unhighlight the button.

What is necessary to keep track the active button? We are assuming whenever we are in the menu then we must be in a button. The active button changes when the mouse moves into a new button. At this time we want to start an instance of the button module running on the new active button. To maintain the active button we have:

```
loop
   ACTIVE_BUTTON := GET_MENU_BUTTON (? MENU, ? MOUSE);
   signal BUTTON (BUTTON), BUTTON_OUT in
      trap CHANGE_BUTTON in
         [
          emit BUTTON (ACTIVE_BUTTON);
          copymodule BUTTON;
          exit THE_END
         ||
          await BUTTON_OUT do exit CHANGE_BUTTON end
         ]
      end
   end
end
```

We first use the GET_MENU_BUTTON function to find the active button. Then we use a *trap-exit* statement for controlling when we move from one button to another within the menu body. This requires running the button module on the active button in parallel with watching for when the mouse leaves this button. When BUTTON_OUT† is received we know that the menu's active button has changed. (When we change the active button, in the same Esterel instant two button modules are running: with BUTTON_OUT the first module terminates and a second begins as the loop restarts.) Note that if ever the button module terminates we terminate the menu-body module by executing the "exit THE_END". Here again we are assuming that the three components of the menu-body module are enclosed in a *trap-exit* construct.

The third component of the menu body's behavior is waiting for mouse-up. Recall that if the module BUTTON terminates then we terminate the menu-body module. There is a button module running whenever we are inside the menu, so there is nothing to do when we are in the menu. In the other case, there is no button module running and thus we "exit THE_END" to terminate the menu-body module.

```
await MOUSE_UP do
   emit AM_I_IN;
   present NO then
      exit THE_END
   end
end
```

5.2. *Special Kinds of Menus: Pop-Ups and Pulldowns*

As we mentioned above the main difference in the various kinds of menus are how one initiates the behavior. For a pulldown, this is done by clicking in what we call the "title" button. For a pop-up, this is done by depressing a specified mouse button. For a fixed menu, one simply clicks inside the menu.

---

† To re-use the button module requires a slight modification. The BUTTON_OUT signal must be declared as an output signal in the external interface (declarations) of the button module. This allows it to be heard by other modules. Otherwise that signal is viewed as local to the button module itself.

To use the menu-body module for a fixed menu one connects the Esterel module so that it runs whenever a mouse button is depressed inside the menu. For a pop-up menu the situation is not more difficult: one must draw and erase the menu-body since it is not always visible on the screen:

```
call DRAW_MENU () (? MENU);
copymodule MENU;
call ERASE_MENU () (? MENU)
```

The declarations for the popup module must, of course, declare the two external functions, DRAW_MENU and ERASE_MENU. Notice that here we are assuming that the mouse and the popup menu are in the same system of coordinates.

In the case of a pulldown menu it is clear that we want to run the pop-up module given above on the pulldown's menu. Since a pulldown menu is activated by clicking in the "title" button, it is natural to assume the mouse and the title button are in the same system of coordinates. Thus we introduce a local signal "MENU_MOUSE" whose value is the mouse coordinates relative to the pulldown's menu.

The behavior for the pulldown has two components running in parallel: one is generating a MENU_MOUSE signal for every MOUSE signal received, the second is running the pop-up module described above.

```
await immediate PULLDOWN;
signal MENU (MENU), MENU_MOUSE (COORD) in
  trap THE_END in
    [
      every immediate MOUSE do
        emit MENU_MOUSE (MENU_COORD (? PULLDOWN, ? MOUSE))
      end
    ||
      emit MENU (GET_MENU_BODY (? PULLDOWN));
      copymodule POP_UP [ signal MENU_MOUSE / MOUSE ];
      exit THE_END
    ]
  end
end
```

Note that for the copymodule of POP_UP we just rename the MOUSE signal to use the local signal MENU_MOUSE.

5.3. *Menubar*

A menubar can be viewed as a grouping of pulldowns in much the same way that a menu is a grouping of buttons. There is again the slight behavioral difference from a row of pulldowns that once one clicks inside one of the title buttons that represents the menubar then it is enough to move into another title button to see the new menu displayed.

The external interface requires the introduction of a type MENUBAR and an input signal of that type. In addition we have input signals for MOUSE and MOUSE_UP and the output signal PERFORM_ACTION. As well we introduce functions for getting the menubar's associated rectangle, a title's associated rectangle, and the current pulldown.

At a high level the behavior of a menubar is similar to that of a menu. That is, one needs to know if one is inside or out of the menubar. When inside the

menubar itself one selects a title button which reveals the corresponding menu. By watching the title buttons we keep track of the current pulldown. This happens while awaiting mouse-up which terminates the behavior of the menubar. Thus we have:

```
< generate inside or out of the menubar >
||
< keep track of current pulldown >
||
< wait for mouse-up then do what's necessary >
```

The first component for the menubar is an instance of the check-rectangle module similar to others we've seen.

Maintaining the current pulldown, however, is not as simple as maintaining the active button of a menu. While it is true whenever the mouse moves within the menubar from one title button to another that the current pulldown changes, this pulldown remains the current pulldown when one is no longer in the menubar.

Thus the code for keeping track of the current pulldown is actually two tasks in parallel. The first watches when the mouse is in the menubar to see if the title button changes. The second runs the pulldown module on the current pulldown. We have:

```
< Maintain current title button >
||
< Run current pulldown >
```

Maintaining the current title button is similar to keeping track of the current button in the menu-body module. That is, when inside the menubar we watch for the mouse to enter a new title button. Each time the title button changes we have a new current pulldown. Once out of the menubar we wait until we enter again. This behavior can be specified as follows:

```
loop
   await immediate MENUBAR_IN;
   do
      < Find current pulldown
        Maintain current title button >
   upto MENUBAR_OUT;
end
```

When we have found the current pulldown, we emit a signal of type PULL-DOWN which will be used by the component which runs an instance of the pull-down module. The code for maintaining the current title button is similar to that of maintaining the active button of a menu, thus we won't go into further detail here.

The component which runs the pulldown module for the current pulldown is, of course, listening to the signals emitted by the code above:

```
var ACTIVE_PULL : PULLDOWN in
   loop
      ACTIVE_PULL := ? PULLDOWN;
      do
           copymodule PULLDOWN;
           exit THE_END
      upto PULLDOWN;
      call ERASE_PULLDOWN_MENU () (ACTIVE_PULL)
   end
end
```

Here we set ACTIVE_PULL to the emitted PULLDOWN signal then run the pulldown module. This module is aborted when a new PULLDOWN signal is emitted (i.e., when the mouse is in a new title button). The ERASE_PULLDOWN_MENU is needed because the aborted pulldown module will not have erased the menu which was drawn by that module. If ever the PULLDOWN module terminates, which it does on mouse-up, then we want to terminate the menubar module.

This completes the maintaining of the current pulldown. What remains is the handling of mouse-up. However, since an instance of the pulldown module is always running and its termination terminates the menubar, there is no special handling for mouse-up.

6. Using the Esterel Code

There are two aspects to interfacing with an Esterel module. The first is the abstract data manipulation performed in that module. What is a button? How to get a button's associated rectangle? etc. The second concerns how one actually uses the code. How does one start the automaton? How to generate an input signal?

For a given Esterel module the user must define the data types and the external functions and procedures. This is typically written in some other host language such as C, Ada, or Lisp. The compilation of Esterel results in the generation of an automaton, a function to call this automaton, and one function for each input signal. To use this automaton one emits an arbitrary number of input signals, by calling the input functions, and then calls the automaton which, updates its state and in turn generates an arbitrary number of output signals. The output signals correspond to functions that the user must also define. Note that all input signals emitted before a call to the automaton are considered simultaneous. One call to the automaton results in one state transition.

The interfacing is complete once the user decides how and when to emit the input signals and when to call the automaton. For example, to use the menu-body module given above, one would like the following situation:

- When one clicks in the menu-body, send input signals MENU and MOUSE and then call the automaton.
- Each time the mouse is moved, send input signal MOUSE and call the automaton.
- When one releases the mouse button, send input signals MOUSE_UP and MOUSE and call the automaton.

It is at this level, and only at this level, one must worry about connecting to any underlying hardware or low-level software.

When one is trying to connect an Esterel module to the outside world, the handling of input and output is very important. Recall the synchrony hypothesis assumes that the Esterel program reacts *instantly* to its input signals by updating its state and generating output signals. This translates practically into being *reasonably* fast.

Thus one must guarantee that emitting signals and external function calls are quick. Input signals are broadcast from the outside world and during the time the automaton is called one must make sure that no other input signals are lost. In our case, we have found execution speed is not a problem for external functions. The time taken by an output signal such as PERFORM_ACTION is dependent on the action performed. Instead of directly performing the action one can note that there is something to do and after the call to the automaton returns do what needs to be done.

7. Related Work and Discussion

In specifying the behavior of graphical objects one must find an appropriate model. We view graphical objects as reactive systems that respond to input events and generate output events. The implementation of such systems as automata (or state machines) is very efficient. However, automata are difficult to design and modifications which are based on concurrency (i.e., the same behavior plus something else happening in parallel) are difficult to make; often one is better off throwing out the existing automaton and starting from scratch.

Esterel is a synchronous programming language designed for implementing reactive systems. It provides parallel constructs that ease programming and maintenance of such systems. An Esterel program is compiled into an automaton making for an efficient implementation. Currently, the automata can be generated in either C or Lisp. Esterel induces a programming style that promotes modularity and limits runtime testing. Also it provides a certain degree of reusability or hierarchy for behaviors, for example, the specification of a menu reuses that of a button.

Ours is not the first attempt in this direction. The "Squeak" language introduced by Cardelli and Pike [5] works along similar lines. In Squeak channels exist as a method for communicating between various processors. Squeak is asynchronous however and is somewhat restrictive in its notion of timing. Recently Hill [11] has introduced the "event response language" (ERL) as a method of encoding concurrent activities. This is a rule-based language where the user specifies conditions and flags that must hold for certain actions to be triggered. The flags are essentially encoding the state of the system, the automaton, but the user is responsible for generating these local variables. Modifying such a specification can prove difficult. Also there is currently no modularity in ERL.

While high-level parallel languages, such as Modula [13] and Ada [1] offer constructs that ease the programming task there is usually some execution overhead to be paid. Also such languages are usually nondeterministic and an important property of reactive systems is their determinism.

We find that concurrency and communication through signals permit us to achieve an appropriate abstraction in defining the behavior of graphical objects. The concurrency provides modularity, a separation of the behavior into various tasks. The scheduling of these tasks is easily done in Esterel using the parallel and signal broadcasting. With other parallel languages the scheduling is done by hand and results in the intermixing of separate tasks that happen to be scheduled at the same time. The broadcasting of signals provides reusability, small tasks recombine with added control. An enclosing task can hide signals from a subtask

(a reused module) and the subtask doesn't know who, if anyone, is listening to its signals.

The reuseability of Esterel code is an essential feature. It allows one to build a hierarchy of behaviors and also to provide others who want to write Esterel code a library of simple modules. Of course, this reuseability is at the Esterel code level not at the compiled code level. One cannot link to some already compiled Esterel module.

Another important feature is the quality of the compiled Esterel code. Esterel code is compiled into automata that are very small† and very efficient. Our graphical objects are used in the Centaur [4] interface (a generic interactive environment system) and the performance is good.

8. Conclusion and Future Work

We have presented here how one can use Esterel to specify the behavior of graphical objects. We believe that the reactive systems model is correct for such interface components. Esterel permits one to describe behaviors at an abstract level. Thus a surprisingly complex task is now much easier. Since our specifications are not dependent on graphics, they are rather portable. As well Esterel modules give a certain level of re-useability that permits one to build from previous modules. Finally since Esterel code is compiled into automata, the resulting behaviors are extremely efficient.

Thus far we have only concentrated on low-level graphical objects: menus, menubars, scrollbars, etc. We are very encouraged with our results. We have described behaviors without calling specific graphic primitives, without using specific features of a given window manager, without explicitly using low-level device calls. The communication done through signals represent abstract events. We would like to specify more sophisticated and customized objects. We also feel that Esterel could be used to specify the interface of full "applications" rather than singular objects.

Acknowledgements

We would like to thank Gérard Berry for introducing us to Esterel and for many helpful discussions regarding this work. As well we thank both Gérard Berry and Gilles Kahn for proofreading this paper.

References

1. ADA, *The Programming Language ADA Reference Manual*, Lecture Notes in Computer Science, Springer-Verlag, (155), 1983.
2. G. BERRY, P. COURONNE, G. GONTHIER, "Synchronous Programming of Reactive Systems: An Introduction to ESTEREL" *Proceedings of the First France-Japan Symposium on Artificial Intelligence and Computer Science*, Tokyo, North-Holland, October 1986. (Also as INRIA Rapport de Recherche No. 647.)
3. G. BERRY, F. BOUSSINOT, P. COURONNE, G. GONTHIER, "ESTEREL v2.2 System Manuals" Collection of Technical Reports, Ecole des Mines, Sophia Antipolis, 1986.

† The menubar's automaton has 10 states and its octal code representation is 1014 bytes.

4. P. BORRAS, D. CLEMENT, T. DESPEYROUX, J. INCERPI, G. KAHN, B. LANG, AND V. PASCUAL. "CENTAUR: the system", *Proceedings of ACM SIGSOFT '88: Third Symposium on Software Development Environments*, Boston, November 1988. (Also as INRIA Rapport de Recherche No. 777).
5. L. CARDELLI AND R. PIKE, "Squeak: a Language for Communicating with Mice", *Proceedings of SIGGRAPH* **19**(3), San Francisco, 1985.
6. L. CARDELLI, "Building User Interfaces by Direct Manipulation", Research Report # 22, DEC Systems Research Center, October 1987.
7. J. CHAILLOUX, ET AL. "LeLisp v15.2:Le Manuel de Référence, INRIA Technical Report, 1986.
8. D. CLÉMENT AND J. INCERPI, "Graphic Objects: Geometry, Graphics, and Behavior", *Third Annual Report*, Esprit Project 348, December 1987.
9. D. CLÉMENT AND J. INCERPI, "Specifying the Behavior of Graphical Objects Using Esterel", INRIA Rapport de Rercherche No. 836, April 1988.
10. M. DEVIN ET AL., "Aida: environnement de développemnt d'applications", ILOG, Paris, 1987.
11. R. HILL, "Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction — The Sassafras UIMS" *ACM Transactions on Graphics*, **5**(3), July 1986.
12. MACINTOSH TOOLKIT Apple Computer Corp.
13. N. WIRTH, *Programming in Modula-2*, Springer Verlag, 1982.
14. X11 TOOLKIT MIT project Athena, February 1987.