Specifying the behavior of graphical objects using Esterel

Dominique Clement, Janet Incerpi

INRIA

UNITÉ DE RECHERCHE
RIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N°836

SPECIFYING THE BEHAVIOR OF GRAPHICAL OBJECTS USING ESTEREL

Dominique CLEMENT
Janet INCERPI

AVRIL 1988

Specifying the Behavior of Graphical Objects Using Esterel

Dominique Clément and Janet Incerpi

Abstract

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.

Définition du comportement d'objets graphiques avec Esterel

Résumé

La définition du comportement d'objets graphiques tels que menus, scrollbars, etc., n'est pas une opération très aisée. Cela est dû en particulier à la présence simultanée d'informations provenant du clavier et de la souris. En conséquence il n'est pas seulement difficile de définir le comportement d'un objet mais aussi de le faire évoluer. Les objets graphiques sont considérés ici comme des systèmes réactifs recevant des signaux et générant des signaux après modification de leur état. Nous présentons ici comment le langage Esterel permet d'obtenir de tels systèmes à partir de spécifications claires, efficaces, et modulaires. Le langage Esterel fournit aussi la possibilité de réutiliser des spécifications déjà existantes.

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 the underlying hardware (or low-level software) interfaces with such events. Here we present how one can specify cleanly and efficiently the behavior of graphical objects by using the Esterel programming language.

Graphical objects are the building blocks of user interfaces. Graphical objects include buttons, menus, scrollbars, menubars, browsers, etc. There are many systems available for using and combining these objects ([8],[12],[10]). These systems allow the user to "design" new objects. This may include specifying layout when combining existing objects or specifying the appearance of a new object but for specifying the behavior of new or existing objects the user is on his own. We feel that specifying the behavior of such objects is a complex task. It may require handling multiple input devices and often the notion of time is important. Even if one takes the basic objects for granted –but where to draw the line– developing higher level objects or dialogs is still a problem. What is the correct model for specifying their behavior?

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. While high-level parallel languages, such as Modula [11] 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. Esterel [2] 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 [4] 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 [9] 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.

Thus far we have only specified the behavior of low-level 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 code is portable, not dependent on an underlying window system. The resulting automata are very efficient. More work on connecting the behavior to the graphical objects is, however, necessary.

We begin by a short presentation of Esterel in the following section. This can be skipped by those readers 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 4, we present more complex examples highlighting the various aspects in specifying behavior. We then explain how one interfaces the generated automata with his system and the problems encountered. Finally we close with a discussion of future work.

2. 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 form appropriate for a particular problem.

2.1. *Programming with Esterel*

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), procedures 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.
- 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's body 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].

2.2. *Interface Concerns*

As mentioned above, we consider graphic objects as reactive systems. Typically the input signals correspond to keyboard and mouse events, while output signals are actions performed by the objects. Timing also plays a role in objects. For example,

† These can be used to handle time and synchronization.

double mouse-click and trill (i.e., auto-repeat) interactors can be easily programmed.

The use of signals allows one to separate the behavior from the graphics and hardware. Thus the behavior is in no way dependent on the underlying window system. Although we eventually must worry about this hook up, it is cleaner to program on a higher level. Also we find that the parallel construct is very natural for graphical objects.

Efficiency is always a concern for interface components. We find that the generated automata are small and efficient. One could directly hand-code the automata but we feel this is a difficult task even for small automata. As well one wins in maintainability with Esterel. This is again evident with the reuse of modules: a button module can be re-used in defining the menu module, etc.

3. 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 and introduces another Esterel module, *check-rectangle* that is useful in specifying many graphical objects.

For every Esterel module it is necessary to have some input and output signals. The graphical object whose behavior we are describing is given as an input signal. Also there are input signals corresponding to mouse events (e.g., releasing mouse buttons, dragging the mouse, etc.) and possibly for handling timing (e.g., clock ticks for auto-repeaters). The output signals we emit are to request that some action be performed (e.g., the action associated with the trigger button).

3.1. *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.

As we mentioned above each module has a declaration part which specifies the external interface for that module. It includes the declaration of user defined types, external functions and procedures, and the input and output signals of the module. For the button module we have the following declarations:

```
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 of type BUTTON, MOUSE of type COORD, and MOUSE_UP (a pure signal). It has but one output signal: PERFORM_ACTION which is of type BUTTON.

We assume the button can supply, through the external function GET_RECTANGLE, 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. Note that by working with the button for the XOR function and the PERFORM_ACTION signal we allow for things like "dormant" buttons (i.e., a button which is currently not active). This is in keeping with an object-oriented programming style for the objects themselves. In this case, the state of the button can be checked to see if it is active before doing what's called for.

The *relation* section gives information about the relationships among the input signals. The first states that the signals BUTTON and MOUSE_UP 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.

† Procedures in Esterel have two argument lists: the first is for call by reference arguments, the second for call by value.

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

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. In the next section we describe the check-rectangle module whose task it is to say whether the mouse is in or out of some object's associated rectangle. For now we assume that there exists such a module that is generating "in" and "out" signals that are heard by the other parallel tasks.

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 parallel task mentioned earlier emits the signals BUTTON_OUT and BUTTON_IN.

For handling mouse-up one equally needs to know whether the mouse is inside the button or not. It is possible to write this component using only the BUTTON_IN and BUTTON_OUT signals. Here again we have an infinite loop, although slightly more complicated: if while waiting for the mouse to be out of the button mouse-up occurs then handle it and exit; if while waiting for the mouse to be inside the button mouse-up occurs just exit. This can be presented as follows:

```
loop
   do
      < when mouse up perform action then exit >
   upto BUTTON_OUT;
   do
      < when mouse up just exit >
   upto BUTTON_IN
end
```

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

The *do-upto* construct executes the statement body until the signal, here either BUTTON_OUT or BUTTON_IN, is heard. When the signal is received the *do-upto* aborts and execution of the next statement begins. In both cases we want to wait until the MOUSE_UP signal is seen and then do something. Thus the *await* construct is just what is needed to finish off this component. In the case where we perform the action we have:

```
await immediate MOUSE_UP do
    call XOR () (? BUTTON);
    emit PERFORM_ACTION (? BUTTON);
    exit THE_END
end
```

When there is no action to perform we simply exit:

```
await immmediate MOUSE_UP do
    exit THE_END
end
```

We should mention two technical details with respect to the use of "immediate" and the "exit" statements. The *immediate* implies that if the awaited signal is already present when the test is made then one executes the code. Thus if one of the enclosing *do-upto* statements aborts (because either a BUTTON_IN or BUTTON_OUT is present) and in the same instant a MOUSE_UP is present then we want the appropriate code executed (without the "immediate", execution would wait for a later MOUSE_UP signal). Otherwise one would wait for the next occurrence of the MOUSE_UP. 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.

We find this handling of mouse-up somewhat awkward, but it represents a first attempt if one can only listen for BUTTON_IN and BUTTON_OUT signals. (Recall we assume we have a module generating such in and out signals.) There are other possibilities even with this assumption. One alternative is to have one component update a variable that tells whether one is inside or not and tests this variable on receiving mouse-up. This is hardly satisfactory either. What we would really like is to query the module which is generating the in and out signals at the time when mouse-up occurs to find out the current state. We return to this after we present the check-rectangle module.

3.2. *Check-Rectangle Module*

We want to specify a module that generates in and out signals in response to mouse displacements with respect to a given rectangle. This module is clearly useful for many graphical objects.

The external interface for the check-rectangle module is rather straightforward:

```
module CHECK_RECTANGLE :
type COORD,
     RECTANGLE;
function IN_RECTANGLE (RECTANGLE, COORD) : boolean;
input MOUSE (COORD),
      CHECK_RECTANGLE (RECTANGLE);
relation CHECK_RECTANGLE => MOUSE;
output IN, OUT;
```

We have two input signals: CHECK_RECTANGLE of type RECTANGLE and MOUSE of type COORD. The two output signals, IN and OUT, are pure signals. The only external function is IN_RECTANGLE which returns true or false given a rectangle and a coordinate (or point). Notice that every CHECK_RECTANGLE signal implies that a MOUSE signal is present.

To start we have a check-rectangle module with the following simple behavior: wait for a rectangle then loop forever signalling whenever the mouse moves in or out of this rectangle. For demonstration's sake, assume for check-rectangle we are initially in the rectangle.

```
await immediate CHECK_RECTANGLE;
loop
   emit IN;
   < terminate when you move out >;
   emit OUT;
   < terminate when you move in >
end
```

Note that awaiting the signal CHECK_RECTANGLE is necessary to have a rectangle to check (currently modules in Esterel do not take parameters). The idea is to just emit IN or OUT when the situation changes thus the code in between is testing whether this has happened or not. Now consider what happens once emitting IN while waiting to emit OUT. Basically we test with each MOUSE signal received whether we are still in the rectangle:

```
% Inside rectangle waiting to move out
trap IN_TO_OUT in
   every MOUSE do
      if IN_RECTANGLE (? CHECK_RECTANGLE, ? MOUSE) else
        exit IN_TO_OUT
      end
   end
end
```

Note that each time a MOUSE signal is heard we use the function IN_RECTANGLE to test whether the mouse is in the rectangle. If not, then the "else" part is executed. Thus we execute the exit for the surrounding *trap* statement. At this point we would execute the "emit OUT" as we saw above and then using a similar *trap* construct wait until the mouse moves in the rectangle.

Of course, for check-rectangle truly to be useful we cannot assume that we are initially in the rectangle. To handle this we use a local signal FROM_OUT, whose presence implies that the mouse is initially outside the rectangle. The changes to our original loop are minimal: we compute whether we are initially out of the rectangle and then start the loop correctly:

```
await immediate CHECK_RECTANGLE;
< compute FROM_OUT >;
loop
   present FROM_OUT else
      emit IN;
      < terminate when you move out >
   end;
   emit OUT;
   < terminate when you move in >
end
```

Note that the *present* construct permits by-passing the first part of the loop. If FROM_OUT is present then the "else" part is not executed and thus we emit OUT and wait until the mouse moves into the rectangle. Computing FROM_OUT is responsible for emitting the signal when necessary. We can simply test whether the mouse is in the rectangle to start:

```
if IN_RECTANGLE (? CHECK_RECTANGLE, ? MOUSE) else
  emit FROM_OUT
end
```

Thus whenever the mouse is not in the rectangle to start we emit FROM_OUT. Often we know whether the mouse is inside the rectangle to start and we can emit a signal to inform the check-rectangle module rather than having the module do the test. In this case < computing FROM_OUT > would check for outside signals. If there are no signals then it does the check as shown above. However whenever outside signals are provided one assumes these are correct. So we can include two new input signals, CHECK_FROM_IN and CHECK_FROM_OUT and < computing FROM_OUT > is then as follows:

```
present CHECK_FROM_OUT then
   emit FROM_OUT
else
   present CHECK_FROM_IN else
      if IN_RECTANGLE (? CHECK_RECTANGLE, ? MOUSE) else
        emit FROM_OUT
      end
   end
end;
```

The use of *present* statements looks for external signals and emits FROM_OUT if necessary. Otherwise IN_RECTANGLE performs the test. Note that this permits the calling module to emit the appropriate signal if it knows that initially one is always inside the rectangle.

Thus far the check-rectangle module, after waiting for an input rectangle and computing FROM_OUT, loops forever emitting IN and OUT whenever the situation changes. This is useful for many other modules, however, as we saw with the trigger button we may want at some point to query the module to find out the current status. That is, an external module can, by emitting a signal, query to find out if the mouse is in the rectangle; Check-rectangle responds to such a signal by emitting the signal YES or NO. We can view the check-rectangle module as doing two tasks in parallel:

```
await immediate CHECK_RECTANGLE;
< compute FROM_OUT >;
[
  < generate IN and OUT >
||
  < answer queries AM_I_IN >
]
```

Note that the external interface of check-rectangle must now be extended to have a new input signal, AM_I_IN, and two new output signals, YES and NO. The behavior of the query-answering component is again an infinite loop: when a query arrives answer YES as long as the mouse is inside the rectangle, when a query arrives answer NO as long as the mouse is out of the rectangle. To start off correctly we can again use the FROM_OUT signal. In Esterel we specify:

```
loop
   present FROM_OUT else
      do
         < answer queries - YES >
      upto OUT
   end;
   do
      < answer queries - NO >
   upto IN
end
```

To know whether or not one is inside or out of the rectangle we simply listen for the IN and OUT signals that are generated in parallel. Notice that it is not necessary to store the state in a variable. Using the *do-upto* statement we have all the queries answered the same until OUT or IN is received. Since a query comes in the form of a signal, we listen for every such signal and respond by emitting the appropriate response. Thus to respond YES we have:

```
every immediate AM_I_IN do emit YES end
```

The "every immediate" statement implies that whenever the signal AM_I_IN is present an emit signal will be done. The "immediate" is necessary because if one of the *do-upto* statements is aborted (because either an IN or OUT signal is present) and in the same instant a AM_I_IN signal is present then we want that the appropriate YES or NO

signal is emitted. Note that the code for answering queries, like that for generating IN and OUT, never terminates.

3.3. *Button Module Revisited*

Earlier we presented the button module as three tasks running in parallel, we return to this module to show how one uses the check-rectangle module and how one uses the button module itself. We begin by showing how to run the check-rectangle module in parallel with the components for controlling the highlighting and handling mouse-up. We then show how to rewrite the handling of mouse-up by querying the check-rectangle module.

The component to < tell whether inside or out of the button > can be specified in Esterel as follows:

```
emit CHECK_RECTANGLE (GET_RECTANGLE (? BUTTON));
emit CHECK_FROM_IN;
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 the necessary signals: i) CHECK_RECTANGLE signal whose value, given by the external function GET_RECTANGLE, is the sensitive area for the button; ii) signal CHECK_FROM_IN because we are assuming when one starts the button module one is already inside the button. Note that the MOUSE signals received by the button module are also used by check-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.)

It is no longer necessary to listen for BUTTON_IN and BUTTON_OUT signals to handle mouse-up. This component waits for mouse-up then queries 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. In any case we use the *trap-exit* mechanism to terminate the module.

The behavior for the simple button is almost complete. The only remaining problem is to have this Esterel module used again and again, each time one clicks in a button. Here we need a loop whose body is just a copymodule of the button module.

```
every immediate BUTTON do
   copymodule BUTTON
end
```

This button-loop module has the same input and output signals as the button module. Note that with every BUTTON signal which is emitted we simply restart the button module.

3.4. *Discussion*

We now outline how one goes about using this code. This includes what the user must write in terms of supporting code and what the Esterel provides the user for finally attaching the behavior to a graphical object. (We return to this later in section 5.)

The user must define the external data types such as COORD, RECTANGLE, and BUTTON plus two functions for each type, an assignment function and an equality predicate. Note that Esterel only utilizes these functions and therefore really doesn't know anything about the types used in the programs. Clearly, the external functions and procedures must also be written by the user.

Attaching the behavior to a graphical object that is a button only requires that one passes that object as an input signal and that one can manipulate this object via XOR and GET_RECTANGLE. One must also be able to "coordinate" the other input signals MOUSE and MOUSE_UP. Functions associated with the input signals and the automaton itself are provided by the compiled Esterel. Calling the MOUSE_UP input function followed by calling the automaton is equivalent to having emitted the signal MOUSE_UP. The user is responsible for writing any output signal functions. An emission of an output signal translates into a call of its corresponding function.

The coordination of input signals is an important aspect in using the Esterel code. For example, above we gave a simple button loop module which each time it sees a BUTTON signal restarts the button module. This could lead to a problem if a button signal is emitted and another button signal is emitted before mouse-up occurs. The problem is that the first button module is instantly aborted (perhaps the button is still highlighted).

Of course, the issue is how does one connect the Esterel code to the outside world. We must guarantee that a second BUTTON signal won't be seen until a MOUSE_UP

has been received. Depending on the system one may have a filter (or processor) which translates external events into the correct Esterel signals. Also it is possible to have the Esterel code deal with this. Consider the button-loop module as two tasks running in parallel. The first is going to ignore button signals that arrive if there is a running button module. However it initiates the button module when there is no running button module. The second task is running the button module.

To initiate the button module we introduce a local signal REAL_BUTTON. This signal is generated only when there is no button module running.

```
signal REAL_BUTTON (BUTTON) in
  [
   loop
      await BUTTON do emit REAL_BUTTON end
      do
         every BUTTON do emit STUPID_GUY end
      upto MOUSE_UP
   end
  ||
   every immediate REAL_BUTTON do
      copymodule BUTTON
   end
  ]
end
```

Notice that the second task starts a button module every time REAL_BUTTON is emitted. The first task emits this signal when a BUTTON signal is heard but then it ignores all other BUTTON signals until MOUSE_UP is received. (The STUPID_GUY signal is just editorial comment when one tries to start another button without finishing an earlier one.) On mouse-up a new BUTTON signal is awaited before emitting REAL_BUTTON.

4. Reuseability 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. We then show how to breakdown the behavior for objects such as a scrollbar. Finally we present general concerns when trying to specify behavior for graphical objects.

4.1. *A General Menu Module*

We now want to specify the behavior for a menu. 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 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.

There is also the question of what kind of menu 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. 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. Note that since we will use the menu-body module for all types of menus (fixed, popup, and pulldown)

we cannot make any assumption about the initial position of the mouse relative to the menu-body. Thus we have:

```
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. If it is not then there is no current button. Thus we loop waiting to see when the mouse is in the menu body and then once inside maintaining the active button until we move out of the menu. 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 whether the mouse is initially inside or out of the menu body. The call to XOR is necessary because the button module 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 thus unhighlight the button.

What is necessary to keep track the active button? Whenever the mouse is inside the menu body we must loop to see whether the active button changes. 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 are assuming whenever we are in the menu then we must be in a button. To maintain the active button 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.

To use the button module as shown above 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.

The third component of the menu body's behavior is waiting for mouse-up. There are two cases to consider; the mouse is either inside or out of the menu. In the former case we inform the running button module that mouse-up has occurred, while in the latter case we simply terminate the menu module.

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

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.

4.2. *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 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. Thus, we must build a menu loop module, similar to the button loop module shown in the previous section.

```
every immediate MENU do
   copymodule MENU
end
```

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:

```
every immediate MENU do
   call DRAW_MENU () (? MENU);
   copymodule MENU;
   call ERASE_MENU () (? MENU)
end
```

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 menu-body module given above on the pulldown's menu. We can easily access this through the addition of an external function, GET_MENU_BODY. The situation for the pulldown is more complex. Since a pulldown menu is activated by clicking in the "title" button, we feel it is natural to assume the mouse and the title button are in the same system of coordinates. However, our menu-body module assumes that the mouse and the menu-body are in the same system of coordinates.

We solve this by introducing a local signal "MENU_MOUSE" whose value is the mouse coordinates relative to the pulldown's menu. To go from coordinates relative to the title button to those relative to the menu body we use an external function, MENU_BODY_COORD. The external interface for a pulldown module is:

```
module PULLDOWN:
type COORD,
     RECTANGLE,
     BUTTON,
     MENU,
     PULLDOWN;
function GET_MENU_BODY (PULLDOWN) : MENU,
         MENU_BODY_COORD (PULLDOWN, COORD) : COORD;
procedure DRAW_MENU () (MENU),
          ERASE_MENU () (MENU);
input PULLDOWN (PULLDOWN),
      MOUSE (COORD),
      MOUSE_UP;
output PERFORM_ACTION (BUTTON);
relation PULLDOWN # MOUSE_UP,
         PULLDOWN => MOUSE,
         MOUSE_UP => MOUSE;
```

Note that the GET_MENU_BODY takes a pulldown as argument and returns its associated menu. The function MENU_BODY_COORD takes as arguments the pulldown and the mouse coordinates and returns a new coordinate relative to the pulldown's menu.

The behavior for the pulldown has two components running in parallel: one is generating MENU_MOUSE signals for every MOUSE signal received, the second is running the menu-body module described above plus handling the drawing and erasing of the menu-body. The latter task draws the menu body, emits the menu-body before doing the copymodule of menu-body. When the menu-body module has terminated, on mouse-up, one simply erases the menu-body and terminates the pulldown module.

```
await immediate PULLDOWN;
signal MENU (MENU), MENU_MOUSE (COORD) in
  trap THE_END in
    [
     every immediate MOUSE do
        emit MENU_MOUSE (MENU_BODY_COORD (? PULLDOWN, ? MOUSE))
     end
    ||
     var MENU := GET_MENU_BODY (? PULLDOWN) : MENU in
        call DRAW_MENU () (MENU);
        emit MENU (MENU);
        copymodule MENU [ signal MENU_MOUSE / MOUSE ];
        call ERASE_MENU () (MENU);
        exit THE_END
     end
    ]
  end
end
```

Note that for the copymodule of MENU we just rename the MOUSE signal to use the local signal MENU_MOUSE. To use the pulldown module we again make a pulldown

loop module similar to what we did above for the menu-body.

4.3. *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 new type MENUBAR and a signal of that type. As well we introduce functions for getting the menubar's associated rectangle, a title's associated rectangle, and the current pulldown. The complete interface is given below:

```
module MENUBAR:
type COORD,
     RECTANGLE,
     BUTTON,
     PULLDOWN,
     MENUBAR;
function GET_BAR_RECTANGLE (MENUBAR) : RECTANGLE,
         GET_TITLE_RECTANGLE (MENUBAR, COORD) : RECTANGLE,
         GET_MENU (MENUBAR, COORD) : PULLDOWN;
procedure ERASE_PULLDOWN_MENU () (PULLDOWN);
input MENUBAR (MENUBAR),
      MOUSE (COORD),
      MOUSE_UP;
relation MENUBAR => MOUSE,
         MOUSE_UP => MOUSE,
         MENUBAR # MOUSE_UP;
output PERFORM_ACTION (BUTTON);
```

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 easiest component for the menubar is, of course, generating whether one is inside or not. This is simply an instance of the check-rectangle module.

```
emit CHECK_RECTANGLE (GET_BAR_RECTANGLE (? MENUBAR));
emit CHECK_FROM_IN;
copymodule CHECK_RECTANGLE [signal MENUBAR_IN / IN,
                                   MENUBAR_OUT / OUT]
```

As usual we assume that the mouse and the menubar are in the same system of coordinates. Again this seems natural since the behavior is triggered by clicking in the menubar.

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. Recall for the menu there is only a selected (active) button as long as the mouse is inside the menu but for a menubar this is not the case. There is always a current pulldown until mouse-up occurs. In fact, one must move out of the menubar in order to select a button on the current pulldown.

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:

```
var ACTIVE_PULL : PULLDOWN in
   [
     < Maintain current title button >
   ||
     < Run current pulldown >
   ]
end
```

The variable ACTIVE_PULL contains the pulldown which corresponds to the currently selected title button. Thus it is the first task that knows which is the active pulldown while the second runs an instance of the pulldown module for this 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:

```
% Assuming:
% – if you're in the menubar then you're in a title button
loop
   await immediate MENUBAR_IN;
   do
      < Find current pulldown
        Maintain current title button >
   upto MENUBAR_OUT;
end
```

To find the current pulldown we make use of the external function, GET_MENU. We emit a signal of type PULLDOWN which will be used by the component which runs an instance of the pulldown module. To maintain the current title button we again use an instance of the check-rectangle module. Here again we use a *trap-exit* to see when we leave the current title button.

```
loop
   signal CHECK_RECTANGLE (RECTANGLE), IN, TITLE_OUT, YES, NO,
          AM_I_IN, CHECK_FROM_IN, CHECK_FROM_OUT in
      emit PULLDOWN (GET_MENU (? MENUBAR, ? MOUSE));
      trap CHANGE_TITLE in
         [
          emit CHECK_RECTANGLE
                   (GET_TITLE_RECTANGLE (? MENUBAR, ?MOUSE));
          emit CHECK_FROM_IN;
          copymodule CHECK_RECTANGLE [ signal TITLE_OUT /OUT ]
         ||
          await TITLE_OUT do exit CHANGE_TITLE end
         ]
      end
   end
end
```

The function GET_TITLE_RECTANGLE provides the rectangle associated with the title button that the mouse is currently in. Whenever the mouse moves outside the title button and yet is inside the menubar then we must have a new current pulldown and the loop restarts. Thus with each change of the title button we must find what is now the current pulldown and then watch to see when we enter a new title button.

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

```
loop
   ACTIVE_PULL := ? PULLDOWN;
   do
        copymodule PULLDOWN;
        exit THE_END
   upto PULLDOWN;
   call ERASE_PULLDOWN_MENU () (ACTIVE_PULL)
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. It is the menubar that knows this old active pulldown should be erased before starting a new instance of the pulldown module. If ever the PULLDOWN module

terminates, which it does on mouse-up, then we want to terminate the menubar module. Here again we are assuming an enclosing *trap-exit* around the three components for the menubar behavior.

This completes the maintaining of the current pulldown. What remains is the handling of mouse-up. Is there something for the menubar to do when mouse-up occurs? At all times there is a running instance of the pulldown module and when the pulldown module terminates, the menubar also terminates. Thus there is no special handling for mouse-up that the menubar must do and this component is unnecessary.

With the menubar behavior specified as shown above we found a problem when one exits and re-enters the menubar in the *same* title button. In this case the current pulldown menu is erased and then redrawn; this is not acceptable esthetically. The reason this happens is that the current pulldown has not changed so another PULLDOWN signal is emitted resulting in: i) the pulldown module being aborted and the current pulldown erased; ii) a new instance of the pulldown module being started (redrawing the same pulldown). To correct this problem we would like to put in a check that the new current pulldown is not the same as the old current pulldown.

This requires modifying the component that maintains the current pulldown. This can now be seen as three tasks in parallel:

```
var ACTIVE_PULL : PULLDOWN in
   [
     < Propose pulldowns and maintain current title button >
   ||
     < Run current pulldown >
   ||
     < Check out proposed pulldowns >
   ]
end
```

The first task is only slightly modified: instead of emitting PULLDOWN each time the mouse enters the menubar we emit a new local signal, PROPOSED_MENU, which is of type PULLDOWN. The third task is going to verify that the proposed menu is different from the currently running pulldown before emitting the PULLDOWN signal.

In fact we have the third component occupy itself with keeping track of the active pulldown since it does the check of the proposed menu. Thus the second component is simplified:

```
loop
   copymodule PULLDOWN;
   exit THE_END
each PULLDOWN
```

The third component behaves as follows: emit the active pulldown, whenever a pulldown is proposed then check if it is different. If it is then there is a new active pulldown,

so we erase the old pulldown and emit a new pulldown signal:

```
% Assuming: you are in the menubar to start
loop
   trap CHANGE_MENU in
      ACTIVE_PULL := ? PROPOSED_MENU;
      emit PULLDOWN (ACTIVE_PULL);
      every PROPOSED_MENU do
         < check if new active pulldown >
      end
   end
end
```

The *trap-exit* is used whenever a menu changes; that is a new active pulldown. The check is as follows:

```
if ? PROPOSED_MENU = ACTIVE_PULL else
  call ERASE_PULLDOWN_MENU () (ACTIVE_PULL);
  exit CHANGE_MENU
end
```

The "else" part is executed only when we have a new active pulldown. Thus we erase the old one and then restart the loop (resetting the ACTIVE_PULL and emitting a real PULLDOWN signal). It is interesting to note that this is our first runtime test; all other tests have involved only signals.

To use this menubar module, we need to make it run every time one clicks inside the menubar. For this we make a menubar loop module that does a copymodule of the menubar for every MENUBAR signal received. This is similar to what we have done for both button and menu.

4.4. *Compound Objects*

Thus far we have presented specifications where the global graphical object has a behavior. For example, the menubar module should be attached to the graphical object which is the menubar. For compound objects, which are composed of *subobjects*, it may be more desirable to attach behaviors to different *subcomponents* of the graphical object rather than to the object itself.

Consider for example a scrollbar. As a graphical object, a scrollbar consists of two trill buttons (which incrementally scroll up and down) and a center component. This center component provides an "index" (or thumb) to position oneself in the object that is being scrolled over. Additionally, if one clicks outside of the index one scrolls up (or down) until the index is at the clicked position.

Now with this representation it is clear that the trill buttons have their own independent behavior. It is the action associated with the buttons that is different –it does the scrolling over some object. One could argue that a basic scrollbar doesn't

need trill buttons. But the main question is does one specify the behavior of the center component as one Esterel module or two?

Let's say we want to write one behavior for the scrollbar (where by scrollbar we now mean just the center component). The Esterel module is simply an if-then-else with a test that finds out whether or not the mouse is in the index. Depending on the result, one calls a module to control the index or one call a module to control the area surrounding the index. This is much the same as a dialog box. If the dialog box contains, say, three buttons do we specify the behavior of the box or just attach to each of the buttons the button-loop behavior.

Clearly one can do this either way, depending on how one attaches the behavior to graphical objects. The code generated by separating the behaviors is somewhat smaller. Even if one chooses one specification, it is always best to make modules out of the separate components. This way they can be used for other specifications if need be. Note that the module which would control the "index" of a scrollbar is exactly what is needed for a "slider". A slider is a graphical object which is used to gauge some data. Physical examples of a slider include a thermostat, the sound control on a stereo, etc. A graphical object slider may be used to control the brightness of the screen, for example.

4.5. *Discussion*

Clearly we feel that the check-rectangle module plays a major part in specifying the behavior of many graphical objects. We think it is very important that such a component is not attached (hard-wired) to the underlying graphics or window system when describing behaviors.

With each object we've specified we have always written a "driving" loop function: every immediate BLAH do copymodule BLAH end. This is because to use the Esterel again and again we must restart the module. We would like to define a global loop which can be parameterized but currently such parametrization is not available although it is foreseen in future versions of Esterel. The same type of problem can be seen with the grouping of objects. We group buttons into a menu and pulldowns into a menubar each time the code is similar. Here we think parameterization could also be helpful.

The reuseability of Esterel code is an essential feature. It allows one to build a kind of 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.

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

5. Using the Esterel Code

Using Esterel for specifying the behavior of graphical objects permits one to abstract the behavior from the underlying graphics or window system. Esterel permits us to describe cleanly and easily the behavior and gives a level of portability. Eventually one must attach the behavior to the graphical objects themselves. Here we present how one goes about using the Esterel modules for graphical objects. This includes what the user must write to interface with an Esterel module as well as how the user activates the resulting automaton. Finally, we mention issues regarding the use of Esterel and how we currently use the specifications shown above.

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? We discuss each of these aspects below.

5.1. *The Handling of Data*

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. Note that within our Esterel code the data manipulation is done mainly using the objects. For example, the menu module manipulates ACTIVE_BUTTON which is retrieved using an external function, GET_MENU_BUTTON. Even for the menu one receives a signal whose value is a menu object. For user defined data types, such as BUTTON, one must also provide two functions: an assignment function, _assign_BUTTON, and a test for equality function, _eq_BUTTON. These are needed when one has a signal or variable of a specific type or when one has a test inside the code itself.

5.2. *Mapping Events to Esterel Signals*

The compilation of Esterel results in the generation of an automaton. To use this automaton one emits an arbitrary number of input signals and then calls the automaton which, updates its state and in turn generates an arbitrary number of output signals. 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. That is, one does not call the automaton associated with a module just once. One emits some input signals, calls the automaton (perhaps some output signals are generated), emits some more input signals, calls again the automaton, etc.

The compilation process generates a number of functions that allows one to emit input signals and to call the automaton. Compiling an Esterel module, M, generates an automaton that can be called by a function of the same name. For each input signal,

S, compilation produces a function, *I_S*, that is to be used each time one wants to emit the signal *S*. For output signals it is the inverse situation, the Esterel code (not the user) emits these signals and it is the user who must provide a function that is called at each emission. Thus for each output signal, *S*, the user must write a function *O_S* which the automaton calls each time it wants to emit the signal. For example, for the simple trigger button the user calls, say, I_MOUSE_UP but he must write the function O_PERFORM_ACTION. (Note that for valued signals the values are just parameters to the corresponding functions.)

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 I_MENU and I_MOUSE and then call the automaton.
- Each time the mouse is moved, send input signals I_MOUSE and call the automaton.
- When one releases the mouse button, send input signals I_MOUSE_UP and I_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.

5.3. *Discussion*

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 to being *reasonably* fast. This requires that emitting signals and calling the automaton, and therefore any functions it calls, be 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. This can be done with a simple queue for such signals. In our case, we have found execution speed is not a problem for external functions. However, the time taken by the output signal, PERFORM_ACTION is dependent on the action to be performed. Instead of directly performing the action one can just note that there is something to do and after the call to the automaton returns do what needs to be done. Again this is handled by a simple event queue.

Currently we use the behaviors specified above for graphical objects which are

running under Le_Lisp [6]. Thus we compile the Esterel modules into a lisp code. Our system is event-driven, making use of the Le_Lisp virtual graphics system. This system provides an event queue where the graphics system posts events for mouse and keyboard inputs. It is through this queue that we manage the input and output signals to the Esterel code. For example, a button object must be able to respond to a "down-event", where a mouse button has been depressed inside the button. Its response is a sequence of Esterel input signal calls followed by a call to the associated automaton. Conversely, the perform action signal simply posts a "perform-action-event" to the queue and the buttons responds to this event by performing its associated action. For more details see [7].

6. 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 any underlying graphics system 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 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. An application such as a paint program could be specified which watches for mouse and keyboard input simultaneously.

Acknowledgements

We would like to thank Gérard Berry for introducing us to Esterel and for many helpful discussions and his encouragement in 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.

4. L. CARDELLI AND R. PIKE, "Squeak: a Language for Communicating with Mice", *Proceedings of SIGGRAPH* **19**(3), San Francisco, 1985.
5. L. CARDELLI, "Building User Interfaces by Direct Manipulation", Research Report # 22, DEC Systems Research Center, October 1987.
6. J. CHAILLOUX, ET AL. "LeLisp v15.2:Le Manuel de Référence, INRIA Technical Report, 1986.
7. D. CLÉMENT AND J. INCERPI, "Graphic Objects: Geometry, Graphics, and Behavior", In preparation.
8. M. DEVIN ET AL., "Aida: environnement de développemnt d'applications", ILOG, Paris, 1987.
9. R. HILL, "Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction — The Sassafras UIMS" *ACM Transactions on Graphics*, **5**(3), July 1986.
10. MACINTOSH TOOLKIT Apple Computer Corp.
11. N. WIRTH, *Programming in Modula-2*, Springer Verlag, 1982.
12. X11 TOOLKIT MIT project Athena, February 1987.

APPENDIX – The Esterel Code

$Header: check.strl,v 1.2 88/02/25 11:11:49 jmi Exp $
Check Rectangle Module runs forever. It emits signals IN/OUT depending if you are in a rectangle or not. It only emits such signals at the start of the module or when the situation changes.

```
module CHECK_RECTANGLE :
type COORD,
     RECTANGLE;
function IN_RECTANGLE (RECTANGLE, COORD) : boolean;
input MOUSE (COORD),
      CHECK_RECTANGLE (RECTANGLE),
      CHECK_FROM_IN,
      CHECK_FROM_OUT;
output IN, OUT;
relation CHECK_RECTANGLE => MOUSE,
         CHECK_FROM_IN => CHECK_RECTANGLE,
         CHECK_FROM_OUT => CHECK_RECTANGLE;
```

This module also answers AM_I_IN signals by emitting YES/NO

```
input AM_I_IN;
output YES, NO;
```

Getting started

```
await immediate CHECK_RECTANGLE;
   % Assuming:
   %   -- The rectangle and mouse are in the same coordinate system.
   %      Thus the IN_RECTANGLE function is rather straightforward.
   signal FROM_OUT in
      % Compute initial situation by generating the local FROM_OUT signal.
      % Assuming
      %   -- If CHECK_FROM_IN/CHECK_FROM_OUT is present then no check needed
      present CHECK_FROM_OUT then
         emit FROM_OUT
      else
         present CHECK_FROM_IN else
            if IN_RECTANGLE (? CHECK_RECTANGLE, ? MOUSE) else
               emit FROM_OUT
            end
         end
      end;
      [
         % Generating IN and OUT
         % Only emit a signal each time the situation changes
         loop
            present FROM_OUT else
               emit IN;
               trap IN_TO_OUT in
                   every MOUSE do
                      if IN_RECTANGLE (? CHECK_RECTANGLE, ? MOUSE) else
                         exit IN_TO_OUT
                      end
                   end
               end
            end;
            emit OUT;
```

```
      trap OUT_TO_IN in
         every MOUSE do
            if IN_RECTANGLE (? CHECK_RECTANGLE, ? MOUSE) then
               exit OUT_TO_IN
            end
         end
      end
   ||
      % Generating YES and NO
      % Only emit a signal when AM_I_IN is received (i.e., only when asked)
      loop
         present FROM_OUT else
            do
               every immediate AM_I_IN do emit YES end
            upto OUT
         end;
         do
            every immediate AM_I_IN do emit NO end
         upto IN
      end
   ]
end
```

$Header: button2.strl,v 1.2 88/02/25 11:11:36 jmi Exp $
A simple trigger button : if mouse-up occurs when inside then the button's action is performed.

```
module BUTTON :
type COORD,
     RECTANGLE,
     BUTTON;
function GET_RECTANGLE (BUTTON) : RECTANGLE; % Mouse-sensitive rectangle
procedure XOR () (BUTTON);                   % Does highlighting/unhighlighting
input BUTTON (BUTTON),
      MOUSE (COORD),
      MOUSE_UP;
relation BUTTON # MOUSE_UP,
         BUTTON => MOUSE,
         MOUSE_UP => MOUSE;
output PERFORM_ACTION (BUTTON);
```

Need this output signal when we want to use this module in a menu

```
output BUTTON_OUT;
```

An input button is needed at the start

```
await immediate BUTTON;
signal CHECK_RECTANGLE (RECTANGLE), BUTTON_IN,
       CHECK_FROM_IN, CHECK_FROM_OUT, AM_I_IN, YES, NO in
   trap THE_END in
      [
         % Check the button rectangle
         % Assuming:
         %   -- you are in the button to start
         %   -- the button and mouse are in the same coordinate system
         emit CHECK_RECTANGLE (GET_RECTANGLE (? BUTTON));
         emit CHECK_FROM_IN;
         copymodule CHECK_RECTANGLE [signal BUTTON_IN / IN,
                                             BUTTON_OUT / OUT]
      ||
         loop
            % Controlling the highlighting of the button
            % Assuming:
            %   -- you are in the button to start
            call XOR () (? BUTTON);
            await BUTTON_OUT;
            call XOR () (? BUTTON);
            await BUTTON_IN
         end
      ||
         % On mouse-up see if inside the button rectangle
         % Then unhighlight and perform the associate action
         await MOUSE_UP do
            emit AM_I_IN;
            present YES then
               call XOR () (? BUTTON);
               emit PERFORM_ACTION (? BUTTON)
            end;
            exit THE_END
         end
      ]
     end
end
```

$Header: mbody3.strl,v 1.5 88/02/28 14:32:45 jmi Exp $
A general-purpose menu module.

```
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;
```

An input menu is needed at start

```
await immediate MENU;
signal MENU_IN, MENU_OUT, AM_I_IN, YES, NO,
       CHECK_RECTANGLE (RECTANGLE), CHECK_FROM_IN, CHECK_FROM_OUT in
    trap THE_END in
      [
         % Check the menu rectangle
         % Assuming:
         %   -- the menu-rectangle and mouse are in same coordinate system
         emit CHECK_RECTANGLE (GET_MENU_RECTANGLE (? MENU));
         copymodule CHECK_RECTANGLE [signal MENU_IN / IN,
                                             MENU_OUT / OUT]
      ||
         loop
            % menu works whether you are inside or outside to start
            await immediate MENU_IN;  % Potential wait to enter the menu
            var ACTIVE_BUTTON : BUTTON in
               do          % upto MENU_OUT
                  loop
                     % Maintaining the active button
                     % Assuming:
                     %  -- if in the menu then must be in a button!
                     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   % If button finishes so does menu
                           ||
                              await BUTTON_OUT do exit CHANGE_BUTTON end
                           ]
                        end
                     end
                  end
               upto MENU_OUT;
               % Outside of menu rectangle; unhighlight the active
               % button and back to the main loop to wait for MENU_IN
               call XOR () (ACTIVE_BUTTON)
            end
```

```
      end
   ||
      % When MOUSE_UP occurs outside menu, terminate this module
      await MOUSE_UP do
         emit AM_I_IN;          % The menu rectangle
         present NO then
            exit THE_END        % Finished with mouse-up outside the menu
         end
      end
   ]
 end
end
```

$Header: pull.strl,v 1.2 88/02/28 14:33:28 jmi Exp $
A Pulldown has the title button which when clicked inside causes the menu-body to appear. Once the body appears it runs just like the MENU module.

```
module PULLDOWN:
type COORD,
     RECTANGLE,
     BUTTON,
     MENU,
     PULLDOWN;
function GET_MENU_BODY (PULLDOWN) : MENU,
         MENU_BODY_COORD (PULLDOWN, COORD) : COORD;
procedure DRAW_MENU () (MENU),
          ERASE_MENU () (MENU);
input PULLDOWN (PULLDOWN),
      MOUSE (COORD),
      MOUSE_UP;
output PERFORM_ACTION (BUTTON);
relation PULLDOWN # MOUSE_UP,
         PULLDOWN => MOUSE,
         MOUSE_UP => MOUSE;
```

Here we are waiting for input pulldown.

```
await immediate PULLDOWN;
signal MENU (MENU), MENU_MOUSE (COORD) in
   trap THE_END in
      [
         % Transform mouse coordinates to be relative to menu's body
         % Assuming:
         %    -- the mouse and title are in the same coordinate system
         %       (This is because the pulldown is activated by the title)
         every immediate MOUSE do
            emit MENU_MOUSE (MENU_BODY_COORD (? PULLDOWN, ? MOUSE))
         end
      ||
         % Here we draw the menu-body and then use the MENU module
         % This runs to completion and then we erase the menu-body.
         % Note: we rename MOUSE to MENU_MOUSE since MENU assumes that
         %       the menu and mouse are in the same coordinate system.
         var MENU := GET_MENU_BODY (? PULLDOWN) : MENU in
            call DRAW_MENU () (MENU);
            emit MENU (MENU);
            copymodule MENU [signal MENU_MOUSE / MOUSE];
            call ERASE_MENU () (MENU);
            exit THE_END
         end
      ]
   end
end
```

$Header: mbar2.strl,v 1.4 88/02/28 14:32:35 jmi Exp $
A menubar (a group of pulldowns)

```
module MENUBAR:
type COORD,
     RECTANGLE,
     BUTTON,
     PULLDOWN,
     MENUBAR;
function GET_BAR_RECTANGLE (MENUBAR) : RECTANGLE,
         GET_TITLE_RECTANGLE (MENUBAR, COORD) : RECTANGLE,
         GET_MENU (MENUBAR, COORD) : PULLDOWN;
procedure ERASE_PULLDOWN_MENU () (PULLDOWN);
input MENUBAR (MENUBAR),
      MOUSE (COORD),
      MOUSE_UP;
output PERFORM_ACTION (BUTTON);
relation MENUBAR => MOUSE,
         MOUSE_UP => MOUSE,
         MENUBAR # MOUSE_UP;
```

Need a menubar to start

```
await immediate MENUBAR;
trap THE_END in
   signal PROPOSED_MENU (PULLDOWN), PULLDOWN (PULLDOWN),
          MENUBAR_IN, MENUBAR_OUT, AM_I_IN, YES, NO,
          CHECK_RECTANGLE (RECTANGLE), CHECK_FROM_IN, CHECK_FROM_OUT in
      [
         % Check the menubar rectangle
         % Assuming:
         %   -- you are in the menubar to start
         %   -- the menubar and mouse are in the same coordinate system
         emit CHECK_RECTANGLE (GET_BAR_RECTANGLE (? MENUBAR));
         emit CHECK_FROM_IN;
         copymodule CHECK_RECTANGLE [signal MENUBAR_IN/IN, MENUBAR_OUT/OUT]
      ||
         % Keep track of current pulldown
         % Assuming:
         %   -- if you're in the menubar then you're in a title button
         %   -- you are in the menubar to start
         var ACTIVE_PULL : PULLDOWN in
            [
               % Maintain current title button & Propose Pulldowns
               loop
                  await immediate MENUBAR_IN;
                  do
                     loop
                        signal IN, TITLE_OUT, CHECK_RECTANGLE (RECTANGLE), NO,
                               CHECK_FROM_IN, CHECK_FROM_OUT, AM_I_IN, YES in
                           emit PROPOSED_MENU (GET_MENU (? MENUBAR, ? MOUSE));
                           trap CHANGE_TITLE in
                              [
                                 emit CHECK_RECTANGLE
                                   (GET_TITLE_RECTANGLE (? MENUBAR, ? MOUSE));
                                 emit CHECK_FROM_IN;
                                 copymodule CHECK_RECTANGLE [signal
                                                             TITLE_OUT/OUT ]
                              ||
                                 await TITLE_OUT do exit CHANGE_TITLE end
                              ]
```

```
                        end
                     end
                  end
               upto MENUBAR_OUT
            end
         ||
            % Every time we emit PULLDOWN we start it running
            loop
               copymodule PULLDOWN;
               exit THE_END
            each PULLDOWN
         ||
            % Check out proposed pulldowns
            % Assuming:
            %   -- you are in the menubar to start
            loop
               trap CHANGE_MENU in
                  ACTIVE_PULL := ? PROPOSED_MENU;
                  emit PULLDOWN (ACTIVE_PULL);
                  every PROPOSED_MENU do
                     if ? PROPOSED_MENU = ACTIVE_PULL else
                        call ERASE_PULLDOWN_MENU () (ACTIVE_PULL);
                        exit CHANGE_MENU
                     end
                  end
               end
            end
         ]
      end
   ]
  end
end
```

$Header: bloop.strl,v 1.4 88/02/26 15:25:01 jmi Exp $
A simple loop around a button. Everytime you see a button signal restart the button module.

```
module BUTTON_LOOP:
type COORD,
     RECTANGLE,
     BUTTON;
input BUTTON (BUTTON),
      MOUSE (COORD),
      MOUSE_UP;
output PERFORM_ACTION (BUTTON);
relation BUTTON # MOUSE_UP,
         BUTTON => MOUSE,
         MOUSE_UP => MOUSE;
signal BUTTON_OUT in          % Output signal of BUTTON module but not of this
   every immediate BUTTON do
      copymodule BUTTON
   end
end
```

$Header: mloop.strl,v 1.3 88/02/26 15:26:08 jmi Exp $
A simple loop for a menu that says each time you see a menu signal then run the menu module

```
module MBODY_LOOP:
type COORD,
     RECTANGLE,
     BUTTON,
     MENU;
input MENU (MENU),
      MOUSE (COORD),
      MOUSE_UP;
output PERFORM_ACTION (BUTTON);
relation MENU # MOUSE_UP,
         MENU => MOUSE,
         MOUSE_UP => MOUSE;
every immediate MENU do
   copymodule MENU
end
```

$Header: ploop.strl,v 1.2 88/02/28 15:06:23 jmi Exp $
This is a loop for a pulldown menu.

```
module PULLDOWN_LOOP:
type COORD,
     RECTANGLE,
     BUTTON,
     MENU,
     PULLDOWN;
input PULLDOWN (PULLDOWN),
      MOUSE (COORD),
      MOUSE_UP;
output PERFORM_ACTION (BUTTON);
relation PULLDOWN # MOUSE_UP,
         PULLDOWN => MOUSE,
         MOUSE_UP => MOUSE;
every immediate PULLDOWN do
   copymodule PULLDOWN
end
.
```

$Header: loop.strl,v 1.3 88/02/26 15:25:28 jmi Exp $
A loop calling the MENUBAR module

```
module MBAR_LOOP:
type COORD,
     RECTANGLE,
     BUTTON,
     PULLDOWN,
     MENUBAR;
input MENUBAR (MENUBAR),
      MOUSE (COORD),
      MOUSE_UP;
output PERFORM_ACTION (BUTTON);
relation MENUBAR => MOUSE,
         MOUSE_UP => MOUSE,
         MENUBAR # MOUSE_UP;
every immediate MENUBAR do
   copymodule MENUBAR
end
.
```

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique