# 21 Graphical user interface

This chapter presents a more realistic model for the graphical user interface we introduced in Chapter 6. It is based on the control console of a real medical device, but the same techniques can be applied to any system where the operator uses a pointing device such as a mouse to select items from on-screen windows and menus, and uses a keyboard to enter information into dialog boxes. Such facilities are provided by many software systems in wide use today, for example the X window system.

A graphical user interface is an example of a *state transition system* driven by *events*. This chapter explains how to model event-driven state transition systems in Z, and shows how to illustrate a Z text with a kind of *state transition diagram* called a *statechart*. This chapter also shows how to use Z to express designs that are partitioned into units or *modules* that are largely independent. In Z these units can include both data and the operations that act on it, so they can represent *classes* in object-oriented programming.

## 21.1 Events

A great advantage of a graphical user interface is that it allows the users to choose operations in whatever order makes the most sense to them, it does not force users through a fixed sequence determined by the designers. All operations are always potentially available, although some operations might have to be disabled at certain times.

This is a good match to Z, where we define operations that have preconditions that might not be satisfied in certain states. In fact, the lack of any built-in sequencing construct in Z is a positive advantage for modelling this kind of system. We will not be tempted to make unfounded assumptions about the ordering of operations because there is none. The order of operations cannot be predicted; it is completely

determined by what the user wants to do. Systems that must respond to events from the outside world whose order of arrival cannot be predicted are said to be *event-driven.*

In our graphical user interface, the events of interest are mouse clicks and keystrokes. The core of every operation is to receive an event and handle it somehow. These events occur in a system called *Console* which we will describe in the next section.

$[EVENT]$

*Event*

$\Delta Console$

$e? : EVENT$

Many events are simply ignored (they do not change the *Console* state).

$Ignore \mathrel{\widehat=} Event \land \Xi Console$

## 21.2 Displays and dialogs

Now let's fill in the *Console* state. The console provides several different displays such as those shown in Figures 6.3 and 6.4. The display which is currently visible on the console is an important component of the state because it determines which items appear and which operations are available.

The rest of the state is concerned with the dialog that occurs when the user enters a new value for one of the settings that controls the machine. The user begins a dialog by positioning the cursor over a setting on a display such as shown in Figure 6.4 and clicking on the mouse button. Then a dialog box appears where the user may type a new value (Figure 21.1, compare to Figure 6.4) and click on buttons to accept or cancel the value. The characters that the user types into the dialog box are stored as a string of text in a buffer. Normally a user can select almost any operation at any time, but once the dialog begins it must be completed before any other operation can be selected. So the state includes a mode to indicate when a dialog is in progress.

$[DISPLAY, SETTING, VALUE, CHAR]$

$TEXT == \mathrm{seq}\, CHAR$

$MODE ::= idle \mid dialog$

$BUTTON ::= accept \mid cancel$

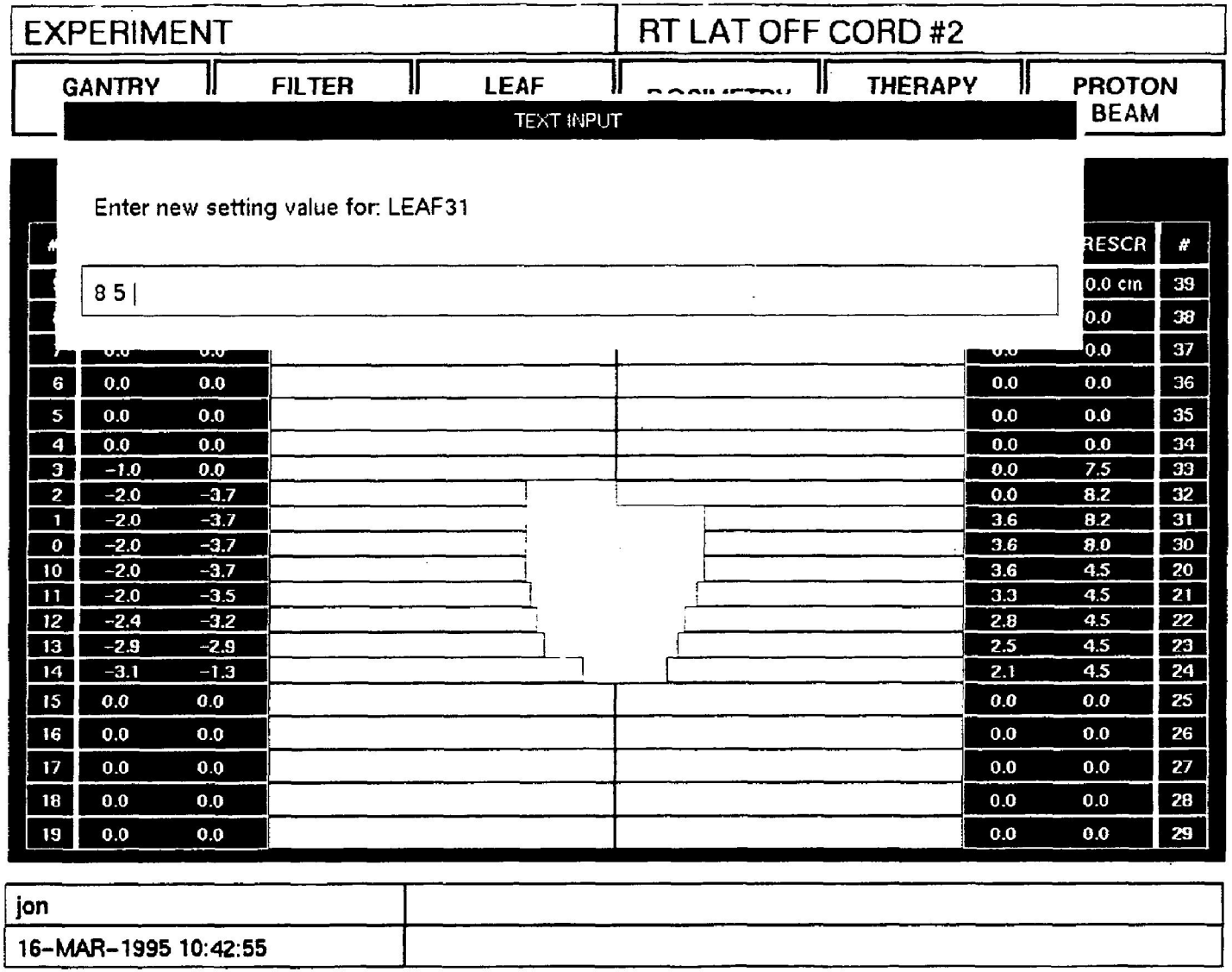


Figure 21.1: Graphical user interface: dialog box.

*Console*

$$display : DISPLAY$$
$$mode : MODE$$
$$buffer : TEXT$$
$$setting : SETTING$$

## 21.3 Selecting a display

The user selects a new display by pressing a function key on the console keyboard (there is a different key for each display). Each time a key is pressed, the *Event* operation occurs. The value of the input event $e?$ tells us which function key was pressed, and that corresponds to the display that the user wants to see. An instance of *EVENT* is actually a data structure that includes an encoded representation of the key.

Here we don't have to provide a detailed description of the event data structure or give directions for decoding the event to determine the function key, and then translate from the function key to the intended display. We hide all of this in a partial function $disp$ that examines the event and returns the display that the user wants to see. The function is partial; events in the domain of $disp$ correspond to display selections.

In Z it is permitted to declare a function without providing its full definition. This supports a top-down specification style where details are deferred.

Here is the operation that selects a new display. This operation is only enabled when the console is not engaged in a dialog.

$$disp : EVENT \pfun DISPLAY$$

*SelectDisplay*

*Event*

---

$$mode = idle$$
$$e? \in \mathrm{dom}\, disp$$
$$display' = disp\ e?$$
$$mode' = mode$$

The predicate $e? \in \mathrm{dom}\, disp$ checks that this event came from one of the display selection keys, and the function application $disp\ e?$ determines which display the

user selected. The screen updates with the new display; this is modelled by $display' = disp\ e?$. In this definition it is necessary to say the mode does not change: $mode' = mode$ (or $mode' = idle$ would mean the same thing). We do not need to say anything about *buffer* and *setting* here because their values do not matter unless a dialog is in progress.

If a dialog is in progress, attempts to select a new display are ignored.

*IgnoreDisplay*

$$Ignore$$

---

$$e? \in \mathrm{dom}\, disp$$
$$mode = dialog$$

We use schema disjunction to combine the two cases.

$$DisplayEvent \mathrel{\widehat=} SelectDisplay \lor IgnoreDisplay$$

The total operation *DisplayEvent* describes everything that can happen when the user presses one of the display selection function keys.

## 21.4 Changing a setting value

Users can enter new prescribed settings at the console. To model this seemingly simple action in Z, we have to break it down into stages and write an operation schema for each stage. In *SelectSetting* the user selects a setting and the dialog begins. Then the user types in the new value; each keystroke invokes *GetChar*. Finally, the user may *Accept* the new value into the machine or *Cancel* the dialog to leave the machine state unchanged. If the user attempts to enter an invalid setting value, the system will not accept it but will *Reprompt*.

### 21.4.1 Starting the dialog

The *SelectSetting* operation occurs when the user clicks on a setting name displayed on the screen. Each such event lies in the domain of the *stg* function. The *setting* state variable records which setting has been selected, as determined by *stg*. A dialog box appears on the screen, and the dialog begins with an empty text buffer (Figure 21.1).

$$stg : EVENT \pfun SETTING$$

*SelectSetting*

$$
\begin{array}{l}
Event \\
\hline
mode = idle \\
e? \in \mathrm{dom}\, stg \\
setting' = stg\, e? \\
mode' = dialog \\
buffer' = \emptyset \\
display' = display
\end{array}
$$

If a dialog is already underway, this operation is disabled.

*IgnoreSetting*

$$
\begin{array}{l}
Ignore \\
\hline
e? \in \mathrm{dom}\, stg \\
mode = dialog
\end{array}
$$

$$SettingEvent \mathrel{\widehat=} SelectSetting \lor IgnoreSetting$$

### 21.4.2 Typing the new value

Each time the user strikes one of the alphanumeric keys, an event occurs, and the *GetChar* operation handles the event. The *char* function extracts the new character from the event, and *edit* updates the buffer. Usually the new character is simply appended to the end, but there might be editing characters that have more complex effects.

$$char : EVENT \pfun CHAR$$
$$edit : (TEXT \times CHAR) \to TEXT$$

*GetChar*

$$
\begin{array}{l}
Event \\
\hline
mode = dialog \\
e? \in \mathrm{dom}\, char \\
buffer' = edit(buffer, char\, e?) \\
mode' = mode \\
setting' = setting \\
display' = display
\end{array}
$$

When the console is not engaged in a dialog, alphanumeric characters are ignored.

*IgnoreChar*

$$\Xi Ignore$$

---

$e? \in \mathrm{dom}\, char$

$mode = idle$

$$CharEvent \mathrel{\widehat=} GetChar \lor IgnoreChar$$

### 21.4.3 Finishing the dialog

When users are finished editing, they may click on an "accept" button in a dialog box (not shown in Fig. 21.1). The system converts the text in the buffer to a value. If the value lies within the range of valid values for the selected setting, it is used to update the prescribed machine settings.

$button : EVENT \pfun BUTTON$

$value : TEXT \to VALUE$

$valid\_ : SETTING \leftrightarrow VALUE$

*Accept*

$$\Xi Event$$

---

$mode = dialog$

$e? \in \mathrm{dom}\, button$

$button\, e? = accept$

$valid(setting, value\ buffer)$

$mode' = idle$

$display' = display$

When the dialog ends, indicated by $mode' = idle$, the dialog box disappears. If the value derived from the buffer contents is not valid, the machine state remains unchanged and the dialog continues. The system may issue a message to reprompt the operator, but we do not model this formally.

*Reprompt*

*Event*

---

$mode = dialog$
$e? \in \text{dom}\, button$
$button\, e? = accept$
$\neg\ valid(setting, value\ buffer)$
$buffer' = buffer$
$mode' = mode$
$display' = display$

Alternatively, the user may cancel the dialog at any time, for example by clicking on a button in a dialog box (not shown in Fig. 21.1).

*Cancel*

*Event*

---

$mode = dialog$
$e? \in \text{dom}\, button$
$button\, e? = cancel$
$mode' = idle$
$display' = display$

Dialog buttons are not available when the console is idle, so these three operations cover all possibilities.

$$ButtonEvent \mathrel{\widehat=} Accept \lor Cancel \lor Reprompt$$

## 21.5 Z and state transition systems

We have used Z to define a system with states and transitions between states: a *finite state machine*. Pictures can help us understand state machines. Figure 21.2 illustrates our system with a kind of *state transition diagram* called a *statechart* [Harel, 1987].

Each bubble in the statechart represents a set of states that satisfies some predicate. In this diagram the outermost bubble represents all *Console* states. The two large bubbles represent the states where $mode = idle$ and $mode = dialog$. The two smaller bubbles represent the states in $mode = dialog$ where the buffer contents represent a valid setting or not: $valid(setting, value\ buffer)$ and $\neg\ valid(setting, value\ buffer)$, respectively. This nesting of state bubbles is one of the innovations that distinguishes

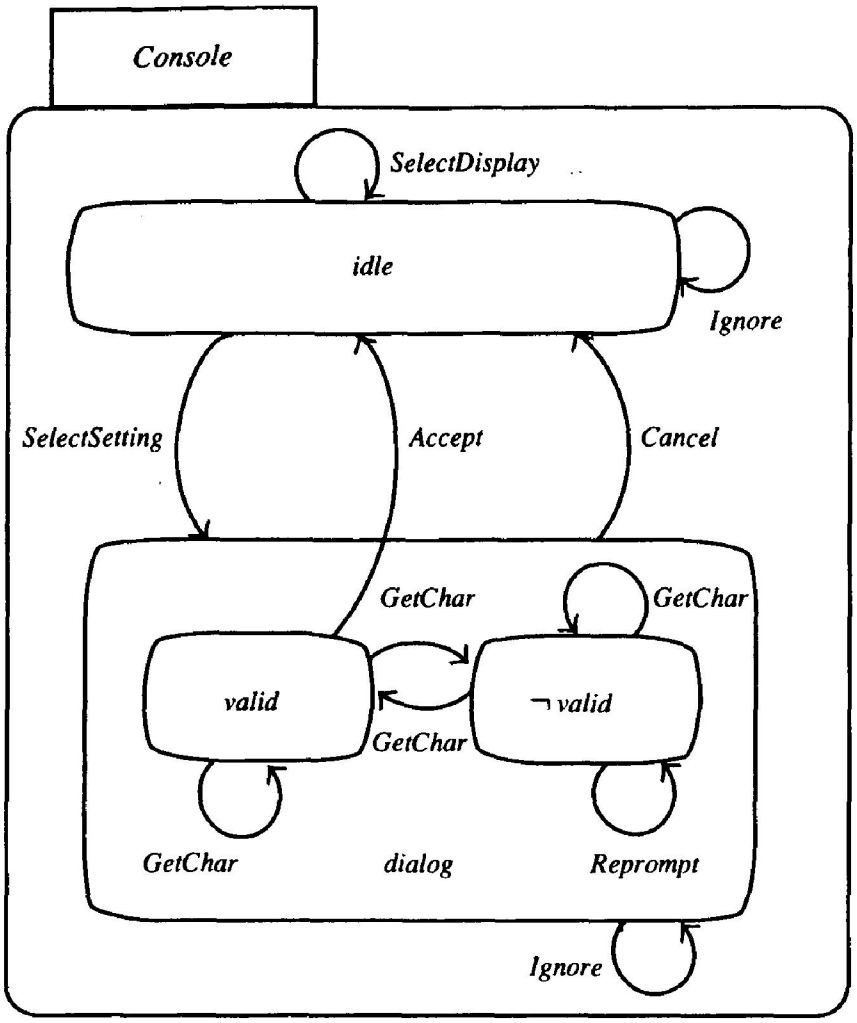


Figure 21.2: Graphical user interface: statechart.

a statechart from an ordinary state transition diagram, such as the one in Figure 6.6. More deeply nested state bubbles correspond to stronger state predicates.

The arrows represent transitions between states, which correspond to Z operation schemas. Each arrow is labelled with the Z operation name, and you can refer to the schemas to find the events that can trigger the transition.

The diagram shows how the operations work together and helps us check for completeness. It is easy to see that we have provided ways in and out of each state bubble and to check that we have dealt with all possible events in every state. However the Z texts contain some essential information that does not appear in the diagram. For example, the *valid* predicate depends on the values of *buffer* and *setting*.

A statechart (or any other diagram) is incomplete; it is always necessary to provide some additional text to explain the diagram. We can use Z to annotate a statechart.

Or, if you prefer to look at it in another way, a statechart can illustrate Z text. The diagram and the formal text complement each other.

## 21.6 Changing the machine state

By itself, a user interface is useless — it has to connect to something else. The purpose of this user interface is to enable the operator to control a machine. We have to model the machine, too.

A good design can be partitioned into units or *modules* that can be described independently. Not just the data, but also the operations can be separated. A unit composed of data and the operations that act on it are abstract data types (ADTs). An abstract data type can model a *class* in object-oriented programming. Our *Console* subsystem and its operations form such a unit. Next we will define the *Machine* subsystem.

The state of the machine is determined by the values of its settings. We don't have to name the settings individually; we can model all of them by a total function from settings to values. In fact there are two functions, one for the *prescribed* settings loaded from a prescription file and another for the *measured* settings read by sensors.

*Machine*

$$measured, prescribed : SETTING \rightarrow VALUE$$

The *NewSetting* operation describes what happens in the *Machine* when a new value $v?$ is assigned to *prescribed* setting $s?$.

*NewSetting*

$$\Delta Machine$$
$$s? : SETTING;\ v? : VALUE$$

---

$$prescribed' = prescribed \oplus \{s? \mapsto v?\}$$
$$actual' = actual$$

We use the override operator $\oplus$ (Section 9.3.2) rather than the equation $prescribed'\ s? = v?$ to express that the values of all the other settings remain unchanged.

Now we can combine the user interface and the machine to make the whole system.

*Sys*

$$Console$$
$$Machine$$

*ChangeSetting* describes what happens to the whole system when the user edits a setting.

*ChangeSetting*

$\Delta Sys$
*Accept*
*NewSetting*

---

$s? = setting$
$v? = value\ buffer$

This is typical Z style for building up operations on whole systems. We use schema inclusion to combine the *Accept* operation from the *Console* subsystem with the *NewSetting* operation from the *Machine* subsystem. The predicate provides the glue that couples the two operations together.

Here $\Delta Sys$ is redundant because all the necessary before and after state variables are declared in *Accept* and *NewSetting*. We include $\Delta Sys$ anyway to make it clear that *ChangeSetting* affects the combined system state.

In fact *ChangeSetting* is the only operation we have defined where the two subsystems interact. We can extend the other operations to make it clear that the machine state is unaffected by most activities at the user interface.

$$SysDisplayEvent \mathrel{\widehat=} DisplayEvent \land \Xi Machine$$

$$SysSettingEvent \mathrel{\widehat=} SettingEvent \land \Xi Machine$$

$$SysCharEvent \mathrel{\widehat=} CharEvent \land \Xi Machine$$

$$SysButtonEvent \mathrel{\widehat=} ChangeSetting \lor (Reprompt \land \Xi Machine) \lor (Cancel \land \Xi Machine)$$

It works the same way in the other direction, too. There are many operations on the *Machine* state that do not involve the user interface. We make the design clean and easy to understand by separating the subsystems. Z nicely supports this modular approach to design.

## 21.7 Conclusions

A Z model like this one can serve as a bridge from requirements expressed in prose and diagrams to code in an executable programming language. The Z texts are more explicit than the informal requirements but they are free of the mass of low-level detail that you have to include to make a program run.

We did not model the appearance of the display or the internals of the event data structure. We did not show how to decode events, how setting values are represented as numbers, or how to test for a valid setting value. We did not describe how to couple *Console* and *Machine* in *ChangeSetting*; we just used schema inclusion (which is essentially logical conjunction) to combine the requirements. The conjunction could be implemented by shared variables or a procedure call with passed parameters. *Console* and *Machine* might even be two different computers communicating across a network. In Z we needn't commit to any of these.

In Z we can defer the details while we focus on essential design issues: partitioning the whole system into subsystems, determining which variables belong in each subsystem, separating required subsystem behaviors into discrete operations, and expressing precisely when each operation is enabled and how it changes the subsystem state.

**Exercise 21.7.1** The console modelled in this chapter does not work quite like the one we described in Chapter 6. Our $SelectDisplay$ operation here allows the user to select any display whenever the console is idle, but in Chapter 6 we described an *Enter* operation that selects one particular display after another in fixed sequence (see Figures 6.6 and 6.7). Define the *Enter* operation that selects the *fields* display when the *patients* display is visible, selects the *setup* display when the *fields* display is visible, and has no effect in other states.