Chapter 7

Jugs example

Practice is the best of all instructors

Publilius

§7 Jugs example

In this section we use an LSD specification to describe an educational application [Townsend]. We then use the insights gained by writing the specification to construct a definitive program for the adm, without the use of the transformation techniques described in §5.4.1. Appendix 8 contains examples based on this chapter. The problem was first considered in [Beynon et al 89a]. The purpose of the chapter is to illustrate the issues which can be addressed using LSD and the adm.

Referring to a Z specification for an interface Wordsworth [1989, p.141] states that "*many questions about the effects of operations can be answered with its help, and the interface can be explored by reasoning about it. Part of the work of constructing it was to draw conclusions from a proposed formal description and compare these conclusions with the informal description. The formal description also helped in formulating questions for the architects...*". In this chapter we show in a similar way that writing an LSD specification given an insight into the synchronisation issues which need to be addressed in the construction of an executable adm program. We attempt to unite the computational power of the adm, as described in Chapter 2, with the cognitive expressiveness of LSD for issues such as asynchronous action and agent perception, as described in Chapter 5. By way of illustration we describe different sorts of user of the educational application developed: these can be thought of as modelling either an "intelligent" pupil who only selects valid options or a pupil who exercises no discrimination concerning option selection, and is only constrained by the characteristics of the system.

The application presents to the pupil a model of a world comprising four elements: two jugs, A and B, each with integral capacity, a source and a sink. The task of the pupil is to achieve a target content of liquid in one of the jugs. Five actions can be performed in the model:

• fill jug A
• fill jug B
• empty jug A
• empty jug B
• pour from one jug into the other, until either the pouring jug is empty or the receiving jug is full

The educational value of the jugs program arises from the number-theoretic properties it demonstrates. In particular any target content of liquid can be obtained precisely when the two capacities are co-prime; otherwise only contents which are multiples of the greatest common divisor of the two capacities can be reached.

We will develop models for the actions of the pupil which are based on the selection of menu options. Suppose the input device consists of a line of five buttons, respectively labelled:

1. Fill A
2. Fill B
3. Empty A
4. Empty B
5. Pour

Each action of the pupil will be either a press or a release of one of these five buttons.

§7.1 LSD specification

We can develop an LSD specification for a pupil who randomly presses and release buttons:

```
agent pupil1() {
protocol
   TRUE § button_i = TRUE, button_i = FALSE ;    (1 ≤ i ≤ 5)
}
```

In this simple specification there is no information perceived by the pupil, who simply presses and releases buttons randomly. No attempt is made, either by internal modelling or external observation of the world, to ensure that only available buttons are pressed, or to press buttons in a goal-oriented manner. The effects of these random, asynchronous button pushes will be constrained by the operational characteristics of the application - the operation of the system cannot be deduced purely from a description of the actions of the pupil.

An alternative formulation for the pupil would involve a modelling of the state of the jugs system by means of external observation. Suppose a composite of the five buttons was displayed on a screen, with valid options highlighted in some manner. We might wish to represent an "intelligent" pupil, who only pressed valid buttons on the input device. Such a pupil would need to take into account the availability of each option, as derived from the screen display:

```
agent pupil2() {
oracle
   avail_i ;      (1 ≤ i ≤ 5)
action
   avail_i § button_i = TRUE, button_i = FALSE ;           (1 ≤ i ≤ 5)
}
```

where elsewhere the following variables are defined:

```
   avail_1 = ¬fullA
   avail_2 = ¬fullB
   avail_3 = ¬emptyA
   avail_4 = ¬emptyB
   avail_5 = avail_6 ∨ avail_7
   avail_6 =  (¬emptyA) € (¬fullB)
   avail_7 = (¬fullA) € (¬emptyB)

   fullA = (contentA == capacityA)
   emptyA = (contentA == 0)
   fullB = (contentB == capacityB)
   emptyB = (contentB == 0)
```

Figure 7.1

This specifies the actions and perceptions of a pupil who acts on feedback about the state of the jugs system. The writing of agent descriptions in terms of their perceptions and capacity to act within the system being modelled is the first stage in going from requirements specification to simulation. Again we have not described the effects of actions, but have merely limited the privilege to act within the system.

In this specification of `pupil2()` we have captured an intent by the pupil to select only valid options. The behaviour of the system which uses the `pupil2()` agent will reflect both the operational constraints imposed by the modelling of the jugs system and the "sensible", non-random behaviour of the `pupil2()` agent. Notice that random button pushes by `pupil1()` may be indistinguishable from the actions of `pupil2()` - we do not distinguish between not being allowed to perform an action and choosing not to do it.

We have written an agent which is capable of monitoring the current state and effecting transitions within that state. There will be many side-effects of user-initiated transitions which the pupil will only indirectly be aware of: changes in the internal state which represents the jug contents, changes in the definitions which affect the screen depiction, if any, of the jugs and their contents, changes in the status of button availability, and so on.

The next stage involves the specification of agents which will respond to the pupil-initiated transitions to effect suitable changes of state to a point where it is appropriate to receive a new menu selection. We might wish to have changes of state effected directly by the agent which observes the button push:

```
agent pour () {
state (int) direction = 0
protocol
   button_1 € avail_1   § contA = |contA| + 1,
   button_2 € avail_2   § contB = |contB| + 1,
   button_3 € avail_3   § contA = |contA| - 1,
   button_4 € avail_4   § contB = |contB| - 1,
   button_5 € avail_6   § direction = 1,
   button_5 € avail_7   § direction = 2,
   (direction == 1) € avail_6
                      § contA = |contA| - 1 ; contB = |contB| + 1,
   (direction == 2) € avail_7
                      § contA = |contA| + 1 ; contB = |contB| - 1,
   (direction == 1) € ¬avail_6
                      § direction = 0,
   (direction == 2) € ¬avail_7
                      § direction = 0
}
```

Another method would involve the dynamic instantiation and deletion of `pour()` agents, with an agent which acts as an interface between the environment (in the form of the pupil agent) and the internal state of the machine. We can also employ definitions to capture the relationship between the contents of the two jugs when the pour operation is requested:

```
agent dialogue-manager() {
oracle (bool) button_i, avail_i       (1 ≤ i ≤ 5)
state (bool) #updating = FALSE
protocol
   avail_i € button_i € ¬updating § pour(i)              (1 ≤ i ≤ 5)
}

agent pour (input) {
oracle
   (bool) avail_i        (1 ≤ i ≤ 7)
state
   (int) contA, contB, updating = true,
         #action = input
derivate
   LIVE = updating
protocol
  (action == 1) € avail_1     §  contA = |contA| + 1,
  (action == 2) € avail$_2$     §  contB = |contB| + 1,
  (action == 3) € avail$_3$     §  contA = |contA| - 1,
  (action == 4) € avail$_4$     §  contA = |contA| - 1,
  (action == 5) € avail$_5$     §  contB = |contA + contB| - contA ;
                                   if avail$_6$ then action=6 else action=7,
  (action == 6) € avail$_6$     §  contA = |contA| - 1,
  (action == 7) € avail$_7$     §  contA = |contA| + 1,
  (1 ≤ action ≤ 4) € ¬avail$_{action}$
                                § updating = false,
  (6 ≤ action ≤ 7) € ¬avail$_{action}$
                                § contB = |contB| ; updating = false
}
```

Figure 7.2

In either case, we will need to introduce other agents to supply the definitions of Figure 7.1, to own the definitions associated with jug characteristics, and so on:

```
agent jugA() {
oracle
   #contA = 0,
   #capA = 4
derivate
   #emptyA = (contA == 0),
   #fullA = (contA == capA)
}

agent jugB() {
oracle
   #contB = 0,
   #capB = 6
derivate
   #emptyB = (contB == 0),
   #fullB = (contB == capB)
}

agent control_panel() {
oracle
   #button$_1$ = FALSE, #button$_2$ = FALSE, #button$_3$ = FALSE,
   #button$_4$ = FALSE, #button$_5$ = FALSE
derivate
   #fullA = (contA == capA),
   #fullB = (contB == capB),
   #emptyA = (contA == 0),
   #emptyB = (contB == 0),
   #avail$_1$ = ¬fullA, #avail$_2$ = ¬fullB,
   #avail$_3$ = ¬emptyA, #avail$_4$ = ¬emptyB,
   #$avail_5$ = $avail_6$ ∨ $avail_7$,
   #$avail_6$ =  (¬emptyA) € (¬fullB),
   #$avail_7$ = (¬fullA) € (¬emptyB)
}
```

We have now specified a complete jugs system. The behaviour of each agent has been described, but we have not addressed the issue of the interaction and synchronisation between agents. We have made the implicit assumption that the agents which effect the change of state dictated by the menu option selected act faster than the pupil agent. All the actions which button pushing causes to happen will therefore occur whilst the button is held down, thus disallowing multiple button pushes and releases whilst one action is occurring. As was discussed in §5.3.2 this assumption cannot be directly represented in an LSD specification. An indirect means of addressing this issue would be to use the updating variable used in Figure 7.2 within the specification of the `pupil2()` agent to describe a third kind of pupil. This would allow the specification of a more intelligent agent, which only presses the next button when no actions are in progress:

```
agent pupil3() {
oracle
   $availbutton_i$        ($1 \le i \le 5$)
state
   updating
action
   $avail_i$ € ¬updating § $button_i$ = true ; $button_i$ = false ($1 \le i \le 5$)
}
```

More complicated interleavings are possible. For instance if the pressing of button 1 is detected by the dialogue manager, but before it can set `updating` to **`true`**, the pupil has released that button and pressed another. Even if the pupil holds the second button, the actions caused by the first button will be those which are carried out. In the next section we construct a program suitable for the abstract definitive machine, using insight gained from the analysis of interaction and synchronisation concerns in §7.1.

§7.2 Adm specification

Synchronisation is required between the actions of the pupil and the reactions of the computer. For instance, it is assumed that the computer is polling the current states of the buttons fast enough that all button presses are detected. Without some means of specifying the relative timings of the agents within the system this is not a valid assumption to make, since it may be that the `pupil3()` agent operates far faster than the `dialogue_manager()` agent, which would result in only a subset of button pushes being detected.

With this issue in mind we construct an adm program which allows neither multiple nor undetected button pushes:

```
entity pupil() {
definition
   down = button1∨button2∨button3∨button4∨button5,
   choice = |rand(5)|, delay = rand(10)
action
   (delay == 1) € ¬down € ¬updating € availchoine
                                                § buttonchoice = true,
   down € updating                              § buttonchoice = false
}
```

The variable `down` records the pupil's knowledge of whether the button is down or not, and the variable `delay` is used to represent the arbitrary nature of her decisions as to when to press the button. This guarantees that:

• no invalid option can be selected

• every button press can be detected

• buttons can only be pressed when all previous actions have been completed

We construct entities to represent the passive elements of the model (i.e. the jugs and control panel) directly from the LSD specification, for instance:

```
entity jugB() {
definition
   contB = 0, capB = 6,
   emptyB = (contB == 0), fullB = (contB == capB)
}
```

We construct the entities which will perform the action selected so as to guarantee that `updating` will become true whenever a button is pressed:

```
entity dialogue-manager() {
action
   buttoni € ¬updating § updating = TRUE ; pour(i) ;      (1 ≤ i ≤ 5)
}

entity pour (input) {
definition
   #action = input ;
action
  (action == 1) € avail1     §  contA = |contA| + 1 ;
  (action == 2) € avail$_2$     §  contB = |contB| + 1 ;
  (action == 3) € avail$_3$     §  contA = |contA| - 1 ;
  (action == 4) € avail$_4$     §  contA = |contA| - 1 ;
  (action == 5) € avail$_5$     §  contB = |contA + contB| - contA ;
                                   if avail$_6$ then action=6 else action=7 ;
  (action == 6) € avail$_6$     §  contA = |contA| - 1 ;
  (action == 7) € avail$_7$     §  contA = |contA| + 1 ;

  (1 ≤ action ≤ 5) € ¬avail$_{action}$
                             § updating = FALSE
  (6 ≤ action ≤ 7) € ¬avail$_{action}$
                             § contB = |contB| ; updating = FALSE    }
```

We have written a program for simulating the jugs model, by using the insight gained from writing the specification in LSD, rather than the transformation process described in §5.4.1. Examples of variations on the program in this chapter are given in Appendix 8, together with examples of their execution.