F1: Extensions and Issues for LSD and EDEN

F1.1. Towards a Better Management of State

This course has focused on the power of definitive scripts to represent state. The connection between state and observation has been emphasised; from this it may be inferred that our attention has been primarily directed at a relatively low-level of abstraction. To convey a complex state, such as the state of the art in definitive programming, it is necessary to do more than draw attention to a single observation – we have to organise a complex body of observations and experiences that contribute to the whole picture. At some level of abstraction, this can be regarded as defining a context for an observation (in the same way that measuring a particular physical parameter may require a complex sequence of actions), but our methods do not make it easy to adopt this perspective.

Many aspects of our research point to the need for higher levels of management of definitive scripts. For instance:

- the use of definitive scripts to represent a single object such as a door is pleasing, but the replication of observations that is needed to make many doors is tedious. An issue to be borne in mind here is that in principle there is nothing to prevent the user of a definitive system that allows easy replication of sets of definitions purporting to represent an object of type T from subverting instances of type T for other ends. For example, after acquiring a door instance, are we to be restricted from redefining it so that it becomes a section of wall (cf permanently locked doors, or bricked up doors). If we want an object-oriented perspective, then we have to introduce some protocols to protect the way in which we use objects within a class. There are clear connections here with the vexing debates in AI about defaults: a penguin is a bird, birds fly, but penguins don't etc.

- the use of definitive scripts in design (a subject that has been addressed by joint work of Cartwright, Beynon and Y P Yung) is attractive not only because of the virtues we have identified in the VCCS, but because of the freedom it allows to accommodate many different design perspectives, and the potential for classification of alternative or tentative designs. It is easy to compile closely related variants of scripts, to interpret auxiliary scripts as paths by which to move from one design script to another, or as alternative versions of particular components. Operations such as freezing scripts, as in creating an icon and thereafter treating as an invariant object, and other more sophisticated ways of associating different privileges with a script according to context, have a role to play in the design process.

- there are many contexts where we would like handle scripts in a more dynamic manner than we have so far successfully conceived

or implemented. For instance, in dynamical systems, or in constraint-maintenance processes, the dependencies may have to be reconfigured dynamically, as when one moving block comes in contact with another, or a constraint is satisfied by an iterative process of propagating updates through a network.

- common experience typically involves exceedingly complicated context switching with respect to interpretation of state. For example, in presenting a mathematical proof, we may say "consider a point p such that .... . *and later* Now let p vary on the locus .... ". In writing a complex computer program, similar activities are involved: we consider the behaviour of program variables first in isolation, then in the context of the program as a whole. For instance, having computed a sequence of values v1, v2, v3, ... in a for-loop, we may express the value of variable v as a function of v1, v2, v3, ... In such a situation, we are moving from contemplation of the state associated with defining vi, for each i, to a state in which all the vi's are defined.

- in modelling in the large, it is essential to deal with the way in which the actions of agents may radically change their privileges. For instance, if we consider passing through a door, rather than merely opening and closing it, we are concerned with modelling an entirely new context. The focus of attention can also switch dramatically in this fashion.

The approach to resolving these issues that presently appears most promising involves introducing abstract operators to describe and compose the states defined by scripts. In terms of an EDEN model of computation, this entails something like defining the symbol table in a definitive fashion (amongst other things). In an LSD framework, it involves embellishing the agent model. One useful concept that can account for dramatic changes in perception is the adoption of different roles. Another possible approach might be to link the presence of agents with that of fragments of system state script that are not directly part of their protocol. Consider for example the fact that, for the schoolboy, the *presence* of the headmaster suppresses certain privileges.

A particularly subtle mode of state representation of interest in this context is Harel's statechart concept. The next section shows how the characteristics of statecharts can be seen to be related to the presence of LSD agents, and the roles that they play.

F1.2. Case Study: The Digital Watch

Harel's statechart model for the digital watch

The statechart for the digital watch can be explained informally as follows.

statechart = state diagrams + depth + orthogonality
+ broadcast communication

Detailed explanation of statecharts to be found in Harel: On Visual Formalisms p522-525.

In the digital watch statechart, have 3 of the 4 characteristic features

state diagrams:
illustrated in update etc

depth:
being in the state of displays means
being in update, or in date, or in stopwatch, or in alarm etc
Will say that being in displays encompasses being in update etc

orthogonality:
light, alarm-st and chime-st are orthogonal components
being in alive state is equivalent to
being in power, main, displays, light, alarm-st and chime-st

[no use is made of broadcast communication.]

Other feature represented in the diagram is the default arrow, which identifies the default state entered when entering an encompassing state.
For example, when watch is first alive:
power is OK, display shows time, light is off,
and alarm & chime are disabled.

Complementary to the default arrow is the "enter by history" transition, which indicates that the state entered when entering an encompassing state will be the state which the system was in most recently. This applies to the stopwatch for instance, which remains in the same mode (e.g. displaying and running) when we return to display it after consulting the absolute time.

An LSD specification for the digital watch

An LSD specification for the digital watch can be developed by pursuing the general philosophy that the pattern of state transitions depicted in the statechart corresponds to the pattern of temporal coincidence between agents. This leads us to introduce agents to correspond to most of the states in the statechart and to model the rest as states within these agents. For instance, the LSD has watch(), displays() and alarm_st() agents, and there is a state variable alarm_st_s within the alarm_st() agent with values D(=1) and E(=2) according to whether the alarm is disabled or enabled. The two characteristics discussed by Harel, associated with depth and orthogonality

respectively, can then be interpreted as corresponding respectively to two methods of describing an agent in terms of simpler agents.

The first way, corresponding to "depth", is associated with defining an agent in terms of the roles it can play: as in

```
a()     =     case a_s of {
                    1: a1();
                    2: a2();
                    ....
              }
```

Here we think of one agent a() as acting through playing one of several roles a1(), a2(), ... At any time a() is in at most one role, so that its state can be seen in Harel's terms as being "playing a1() XOR playing a2() XOR ....".

The second way, corresponding to orthogonality, is associated with defining an agent as a parallel composition of several agents: as in

a() = b() || c() || d() .....

In this case, the agents a(), b(), c() and d() all have exactly the same period of existence. In Harel's terms, the current state of agent a() is defined as being the current state of b() AND the current state of c() AND the current state of d() ... .

As an example of the role construct, we can consider the displays() agent, which either plays the role of the display_current_time() agent or the display_current_chime_time() agent or the display_current_date() agent or the display_current_stopwatch_time() agent etc.

As an example of the parallel construct, we can consider the watch() agent itself, which is defined by the parallel composition of several agents:

main() || light() || alarm_st() || chime_st().

[The omission of the power() agent is deliberate, as explained below.]

In summary, what we have done is to interpret the statechart as a specification of the conditions for liveness of LSD agents. Where there is depth, we think of an agent playing more and more specific roles. Where there is orthogonality, we think of agents operating in parallel.

There is a limitation on the agents that can be defined using the two constructs we have introduced. If a() is the parallel composition of b(), c(), d(), ... then the agents b(), c(0 and d() must have the same period of existence as a() itself. And if a1() is one of the roles played by a(), then a1() must exist for a period of time nested within the lifetime of a().

On the other hand, there are situations in which an agent spawns another agent that has an independent existence, and can have a different period of existence from its creator. The beep() agent associated with the statechart is best conceived as having a lifetime briefer than the watch. When the circumstances that lead to beeping arise, the main() agent, responsible for

display functions, runs a beeping process in parallel, until such time as the beeper is disabled.

There are other examples of hidden agents. The function of the agents within displays() is to ensure that various information relating to time, and times associated with the functions of the watch such as the the alarm, are displayed. What is on the display is a function of the actual time and the current status of displays() (determined by which of its roles it is currently playing). The agents that maintain the values to be displayed are not all represented in the statechart. To model for the passage of time, we must introduce a clock() agent. There must also be a stopwatch component to the watch, complementary to the stopwatch display role, that takes the form of an independent resettable clock running in parallel with the main watch. The current time set for the alarm can conveniently be incorporated into the state of the alarm_st() agent, which is already independent of the alarm time display role in the statechart.

As another refinement, it seems appropriate to separate the power() agent from the watch() itself. Effectively, power is the agent that provides the energy to keep the watch alive, and with an appropriate function of time to represent the battery charge, the watch automatically ceases to function after a certain time has elapsed. The top-level specification then has form:

```
agent power() {
        derivate power_s = battery_charge(time)      // three-valued 2,1,0
}

agent watch() {
    derivate   LIVE = (power_s >= 1)
    oracle  power_s
    agent   main() {                    // as specified below
        .....
    }
    agent alarm_st() {
        derivate    LIVE=LIVE_main
        oracle     LIVE_main , displays_s, alarm_s
        state           alarm_s = D, set_time = 00.00
        handle     alarm_s   // 1:Disab, 2:Enab
        protocol  displays_s == A & alarm_st == D & !d -> alarm_st = E
                        displays_s == A & alarm_st == E & !d -> alarm_st = D
    }                                   // !d here means that button d has been pressed
    agent chime_st() {        // and several other agents, such as light() etc
    }
    agent clock() {
    }
    agent stopwatch() {
    }
    .......
}
```

```
agent main() {
        derivate  LIVE=LIVE_watch
        oracle    LIVE_watch, main_s, alarm_s
        state           main_s = D     //     an integer – 1:Displays, 2:Beep||displays
        handle    main_s
        protocol
            (main_s==D) & (time==set_time) & alarm_s==E -> main_s=B
        agent displays() {
            derivate  LIVE=LIVE_main
            oracle    LIVE_main
            state           displays_s = T //     1:Time, 2:Update, 3:Date, etc
            handle    displays_s, update_s, upalarm_s
            protocol  displays_s==T & !c -> update_s=1,
                            displays_s==T & !d -> displays_s=D,
                            displays_s==A & !c -> upalarm_s=1,
                            .....
                            displays_s!=S & displays_s!=T &2-min -> displays_s=T
            agent disp_date() {
                derivate  LIVE = LIVE_displays & displays_s==D,
                                "watch_display = date as of clock()"
                oracle    LIVE_displays, displays_s
            }
            agent disp_time() { ......
            }
            agent disp_upalarm() {
                derivate  LIVE = LIVE_displays & upalarm_s>0,
                                displays_s = (upalarm_s==0%4)?A:UA
                                "watch_display = time as of alarm setting with
                                                                right digit highlighted"
                oracle    LIVE_displays, upalarm_s
                state           upalarm_s = M
                                // 1:Min, 2:TenMin, 3:Hr, 4=0(mod 4): Alarm
                handle    upalarm_s, set_time
                protocol  !b or 2-min -> upalarm_s=A,
                                !c -> upalarm++,
                                "event -> update set_time so as to
                                                increment highlighted digit in set_time"
            }
            .....................
        }
        agent beep() {
            derivate  LIVE=LIVE_main & main_s==B
            state           main_s
            protocol  beep_stop -> main_s
        }
}       // end main()
```

A SCOUT-DoNaLD-EDEN implementation: suggestions for further work

A rough implementation of the digital watch and statechart animation is specified by the three files: digwatch.s, digwatch.e and watchmech.e. These three files address different aspects of the model:

- digwatch.s is the SCOUT-DoNaLD script for the visualisation of the watch and statechart
- digwatch.e is the EDEN implementation of the agents that control the display functions
- watchmech.e describes the internal functions of the watch that govern what actually happens on the display.

(To confuse matters, the file digwatch.e contains EDEN functions that were used to generate the visualisation files digwatch.s: these could of course be adapted as necessary if a new mode of visualisation were to be adopted.)

The present implementation is currently in a reasonably advanced state of development, but shows signs of unstructured incremental design. There are comments in the program in places, but these can be quite misleading. For instance, the purpose of the function id_ix is to associate a unique integer index with each named state in the statechart. The EDEN function on p3 of digwatch.e (lines 156-163) achieves this by finding the index of an identifier in the symboltable, but is much slower in execution than the alternative function defined at l165. Some of the EDEN code is esoteric, and addresses somewhat tangential issues, such as generation of the SCOUT/DoNaLD script for the visualisation of the statechart; the brief commentary below is intended to help you to focus on the parts of the model that are of most interest. (Note that in its current state of development, the software is somewhere between a model and a program. It is in effect a script, together with a set of agents whose protocols for action have been partly animated via EDEN actions.)

The code is presented as a useful basis for further experiments that might address several different issues, both in respect of written and practical work. For instance:

- intelligibility

  A computer simulation of this complexity would typically be hard to understand and refine. How intelligible do you find the SDE model to be? To what extent is comprehension of the EDEN fragments in digwatch.e assisted by the LSD specification? What do you see as the qualities / defects of the specification?

  How easily can you reverse engineer the model? For instance, can you supply LSD specifications for the missing components, like the stopwatch? Can you figure out the (crucial) role played by the filter action (digwatch.e p10) – is this role represented in the LSD specification outlined above? and, if not, how would you represent it?

  Can you re-organise, document and simplify the program? For instance, can you produce a version of the digital watch simulation as well-organised and documented as that developed by Ian Bridge for the VCCS?

  There is an action called markchart on p6 of digwatch.e. Is it possible to replac this by a piece of definitive script?

  I have claimed that you can gain insight into definitive scripts by interacting with it in an experimental mode e.g. extracting pieces of the script and exercising them in isolation. Can you substantiate this claim?

- nature of the model

  What kind of assurance can you get that the simulation is correct? I have claimed that you can confirm that the simulation works by judicious experiment, and that tracing any anomalous behaviour is relatively easy because of the close correspondence between the variables in the model and meaningful observations of the watch. Can you verify / refute this? Can you supply formal arguments to justify correctness claims for any component?

- modifiability and enhancement

  How easily can you modify the code? Several possible enhancements would be of interest. The present program doesn't label the states of the statechart, nor does it include edges to indicate transitions on button selection. A better interface might replace the DoNaLD visualisation of the statechart by SCOUT windows, and use a colour convention to distinguish the buttons, to indicate which is the current state, and which state would be entered on pressing (say) the red button.

  The model is incomplete (like the statechart) in that the update functions (associated with the update and upalarm states in the statechart) are not specified. Can you add these to the model? What impact would it have on the LSD specification, and the EDEN code? How would it affect the statechart if these functions were introduced?

  There are EDEN procedures set_time() and inc_time() to simulate the effect of running and setting the watch. The digital watch doesn't run in real-time in its present form, but could you link it to the computer system time so that (say) it runs at speed 5 times slower than real-time?

To what extent can you demonstrate the scope for specifying alternative models by slotting in different pieces of script into the existing model? For instance, can you model a watch with an analogue display?

There is a commercial package for manipulation of statecharts that I believe allows them to be constructed by direct manipulation – i.e. by drawing the statechart on the display. How easy would it be to simulate this type of activity on top of the present model?

- Comparative study

  How does the model compare with (say) an OOP implementation? In what respects would (say) a C++ implementation be an improvement? How big would it be? Would it have similar characteristics to the above implementation?

  Can you suggest ways in which EDEN could be enhanced to simplify the specification task? In what respects could the development environment for LSD to SCOUT-DoNaLD-EDEN be improved?

References

David Harel On Visual Formalisms CACM 1988, 31(5), p514-524

Explain the LSD ....

EDEN translation by hand!

Visualisation aspects: object-oriented flavour