F1: Extensions and Issues for LSD and EDEN

F1.1. Towards better Management of State

definitive scripts represent state via observation
=> at low-level of abstraction

convey a complex state via
complex body of observations and experiences

cf context for an experimental observation

Motivation for higher levels of management of definitive scripts

Other areas also motivate script management

- definitive scripts quite good for a single object e.g. door
  but tedious to make many doors

  Why not definitive script to represent class?

  make instance ≠ "acquire definitive script"

  [can subvert instances of type T by redefinition
  as in converting door into a section of wall
  (cf permanently locked doors, or bricked up doors)]

  For object-oriented perspective must impose protocols
  protect the way in which we use objects within a class.

  cf defaults in AI:
  penguin is bird, birds fly, but penguins don't etc.

i.e. OOP = mode of using definitive scripts
"formal" criterion for such modelling being object-oriented
in terms of independent observation of
system being modelled + definitive modeller making script

Also a need for different kind of class-instance relationship:
viz. definitive, where change in class affects instances

cf redesign of car doesn't usually affect cars already produced
change in tax laws does

- definitive scripts for design (cf Cartwright, WMB, YPY)

  incremental animation of the requirement etc (cf VCCS ...)

  but also potentially
  accommodates many different design perspectives
  classify alternative or tentative designs

  e.g. store variants of scripts via
  - scripts as paths mapping one design script to another
  - alternative versions of particular components

Relevant activities in design:
"freezing" scripts (defn -> value)
e.g. designing an icon, then treating as invariant
associating different privileges according to context

- dynamic generation of scripts

  e.g. dynamical system: moving blocks come in contact
  constraint satisfaction through iterative updates

- context switching in the interpretation of state

  In proof presentation:
  "consider a point p such that .... .
  and later "Now let p vary on the locus .... ".

  In programming, define f(v1, v2, v3, ...)
  after computing values v1, v2, v3, ... in a for-loop
  => first consider state associated with vi, for each i,
  then state in which all the vi's are defined

- change in observation can change *context* for observation

  e.g. actions of agents may radically change their privilege. cf *passing through a door*, not just opening and closing
  Focus of attention can also switch dramatically (e.g. alarm)

How to handle need for better management?

Abstract ops to describe / compose states defined by scripts?

In EDEN, suggests defining symbol table definitively

In LSD framework, involves embellishing the agent model
e.g. via the adoption of different roles

In ADM, handle presence / absence of entities definitively
e.g. link presence of agent definitively
to presence of script fragments other than agent protocol
cf *for the schoolboy,*
presence of headmaster suppresses privileges

Harel's statecharts and LSD agents

Harel's statechart concept = subtle mode of state representation

Can relate characteristics of statecharts to
presence of LSD agents
roles that LSD agents play

Idea: evident power of statechart abstraction has principled basis

F1.2. Case Study: The Digital Watch

cf Harel's statechart model for the digital watch

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

The statechart for the digital watch: an overview

statechart = **state diagrams + depth + orthogonality**
+ broadcast communication

cf Harel: On Visual Formalisms p522-525.

Digital watch statechart shows 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

"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, **and** in main, **and** in displays etc

[no use is made of broadcast communication.]

Other features

- default arrow --> 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.

- enter by history transition

  state entered when entering an encompassing state
  = the state which the system was in most recently

  – for example, stopwatch:
  remains in the same mode (e.g. displaying & running)
  when return to display it after consulting current time.

An LSD specification for the digital watch

General idea:

pattern of state transitions depicted in the statechart
<-->
pattern of temporal coincidence between agents

Significance: statechart has a hidden agent-oriented basis

Construct LSD specification thus:

introduce
agents <--> non-composite states in the statechart
model the rest as states within these agents

=> watch(), displays() and alarm_st() become LSD agents
and is state variable alarm_st_s in alarm_st() agent
with value D(=1) or E(=2) ≡ alarm is **d**isabled or **e**nabled

Depth and orthogonality <--> two kinds of agent decomposition

depth ≡ defining an agent via *the roles it can play*

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

Agent a() acts in one of several roles a1(), a2(), ...
At any time a() is in **at most one** role
"playing a1() XOR playing a2() XOR ....".

orthogonality
≡ defining an agent via *parallel composition of subagents*

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

Agents a(), b(), c() & d() all have **same** period of existence

Current state of agent a() is defined by
(current state of b() , current state of c() , current state of d() ... .)

Example of role construct:

displays() agent plays the role of
display_current_time() agent
or display_current_chime_time() agent
or display_current_date() agent
or display_current_stopwatch_time() agent etc.

Example of parallel construct:

watch() agent = parallel composition of several agents:
main() || light() || alarm_st() || chime_st().

[NB deliberate omission of the power() agent]

Summary

can interpret statechart as

specifying conditions for liveness of LSD agents

depth ≡ an agent playing more and more specific roles
orthogonality ≡ agent represents agents operating in parallel

Constraints

a() = parallel composition of b(), c(), d(), ...
=> agents b(), c(0 & d() have same lifetime as a()

a1() is one of the roles played by a()
=> lifetime of a1() is nested within the lifetime of a()

Comment on statechart abstraction

Actually depth and orthogonality not as general as LSD agents

– in LSD, an agent can spawn another independent agent
with unrelated period of existence from its creator

e.g. beep() agent in the statechart best conceived as

an **independent** beeping process
running in parallel with main() agent
[responsible for display functions]
until beeper is disabled.

Other hidden agents

function of agents within displays() to specify
nature of the info that is currently being displayed
[e.g. the watch is currently displaying "the alarm setting"]

BUT
what *actually* appears on the display
= function of the actual time
internal state of watch (e.g. memory of alarm setting)
and current status of displays()

AND
the agents that maintain the actual values to be displayed
are not all represented in the statechart

To model these need
- a clock() agent
- a stopwatch component
to complement the stopwatch display role

[NB stopwatch continues to function when NOT displayed]

Cf current time set for the alarm:
can be incorporated into the state of the alarm_st() agent – which is already independent of alarm time display role

Also separate power() agent from watch() itself:

power = agent that provides energy to keep the watch alive

use derivate of time to represent the battery charge
=> watch ceases to function after a certain time lapse

A SCOUT-DoNaLD-EDEN implementation

Suggestions for further work

Rough implementation of digital watch & statechart animation:

Three files: digwatch.s, digwatch.e and watchmech.e

- digwatch.s = 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 is EDEN code that
  describes the internal functions of the watch

NB digwatch.e contains the EDEN functions that were used to
generate the visualisation files digwatch.s

Advanced state of development
BUT unstructured incremental design

[Caution!
Comments can be quite misleading: .NB id_ix
esoteric code for visualisation
not complete program, but partly animated script]

Possible topics for written and practical work:

- intelligibility

  A computer simulation of this complexity normally hard to understand and refine.

  How intelligible do you find the SDE model to be?
  How helpful is LSD specification in interpreting digwatch.e?
  What are 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?

  If not, how would you represent it?

  Can you re-organise, document and simplify the program?

  cf Ian Bridge's VCCS!

  There is an action called markchart on p6 of digwatch.e.
  Can you replace this by a piece of definitive script?

  I have claimed that you can gain insight by experimenting with extracts the script in isolation.
  Can you substantiate / refute this claim?

- nature of the model

  How sure can you be that the simulation is faithful?

  I have claimed that
  • you can confirm faithfulness by judicious experiment
  • tracing anomalous behaviour easy, since
  variables in the model <--> observations of real 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?

  Possible enhancements

  - label the states of the statechart,
  - add edges to indicate transitions on button selection. • better interface: SCOUT buttons + colour convention

  - add update functions for alarm and time
    (<--> update and upalarm states in the statechart)

  Consider impact on LSD specification / EDEN code
  cf affect on statechart if these functions introduced.

  set_time() & inc_time() simulate running & setting the watch

  Simulation digital watch doesn't presently run in real-time.
  Could you link it to the computer system time so that (say) it runs at speed 5 times slower than real-time?

  Can you show scope for specifying alternative models by slotting in different pieces of script into the existing model? E.g. can you model a watch with an analogue display?

  Statemate is a commercial tool for manipulating statecharts that allows you to construct them via a GUI.
  How easy would it be to simulate this type of activity?

- Comparative study

  Compare EDEN with (say) an OOP implementation?
  How would (say) a C++ implementation be better?
  How big would it be?
  How far would it resemble the EDEN implementation?

  How could EDEN be enhanced for the specification task?

  In what respects could the development environment for LSD to SCOUT-DoNaLD-EDEN be improved?

Explain the LSD ....

EDEN translation by hand!

Visualisation aspects: object-oriented flavour