8

Agent-Oriented Definitive Programming

The central concept of definitive programming is modelling a state by a set of definitions. The expressive power of definitions is enhanced by the techniques we have built up for integrating several definitive notations. But whatever descriptive power a set of definitions has, it is meant to describe only one state. For a typical programming task, it is insufficient to have states without transitions.

EDEN procedures and actions can program the transition between states, but EDEN's control structure is fundamentally imperative. This thesis claims that definitive programming is an exploratory programming paradigm. We would like to see that a definitive program is adaptable to the RUDE cycle of software development. A definitive program should be efficient to *Run*, easy to *Understand* and *Debug*, and on-line *Editable*. We have shown in the previous chapters that definitive states fulfil these requirements, and we would like to see the transition control structure over definitive

states exhibits similar advantages. Although EDEN is the best developed software tool for definitive programming so far, its imperative control structure makes it a poor choice for specifying transitions.

This chapter introduces an agent-oriented approach to transition control of definitive states. The LSD agent-protocol specification language was first proposed by Beynon and Norris in [Beynon86b, BN87] and had its major development by Slade [Slade89]. The purpose of the LSD language is to specify the privileges of the agents to act upon one another in a system. An LSD specification describes an essential part, but not all, of the behaviour of the agents. For this reason, an LSD specification is not an executable program, and it is not expected to be executable. This chapter reviews the LSD programming environment, describes its development since 1987 and gives suggestions for its future development.

LSD was developed for concurrent programming; it was also designed to model agent activities. Therefore, a program derived from an LSD specification should be efficient to run; an LSD specification should be easy to understand and debug. However, the prototyping facility for animating an LSD specification is not as convenient as it might be. The animation of an LSD specification relies on a programming tool, namely the ADM definitive language [BSY88]. The current transformation process from an LSD specification to an ADM program has a few limitations:

1. An LSD specification has an ambiguous interpretation but an ADM program has unambiguous operational semantics. The transformation from LSD to ADM requires additional information about what we have called *scenario information* or *simulation decisions*. This information is not part of an LSD specification. For this reason, the first problem of the transformation is that it cannot be automated.

2. Since the ADM language has only the integer data type, the LSD specifications that can be transformed into an ADM program are just those that require integer, or

more generally enumerated data types. This limits the range of application LSD can specify.

3. The only ADM output channel is a print-statement. ADM is not an ideal environment for the visualisation of the current definitive state.

Our research target in connection with LSD is therefore to seek ways of improving the usability of LSD. To this end, my contribution in this thesis has been:

1. to evaluate and suggest improvements for the LSD notation. These suggestions should make LSD more expressive, and at the same time, preserve the essential characteristics of LSD.

2. to write an ADM-to-EDEN translator. By using automatic translation of an ADM program to an EDEN program in conjunction with other definitive scripts for graphical presentation, a graphical animation of the ADM program can be obtained using the Scout system.

Figure 8.1 shows schematically how we may currently animate an LSD specification. The sample outputs in Figure 8.1 and 8.2 are extracted from a worked railway station simulation example.

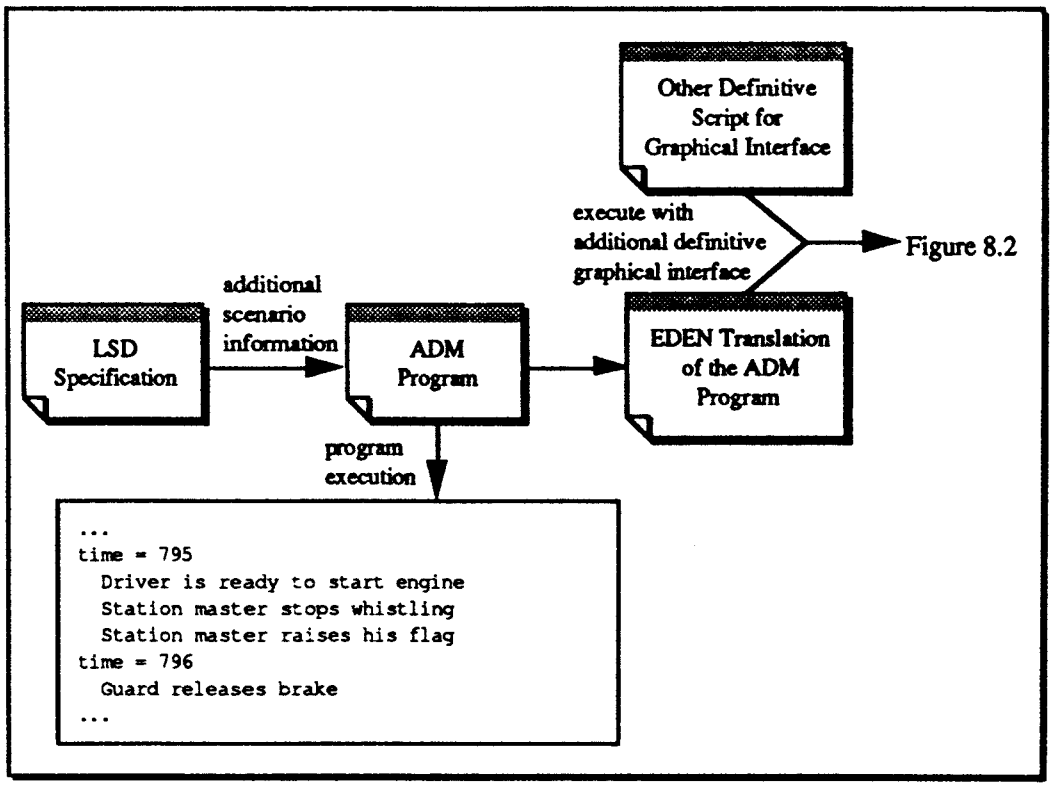

Figure 8.1: Procedures for Animating an LSD Specification

Figure 8.1 should be interpreted as follows. An LSD specification is first transformed into an ADM program. The transformation process involves decision making for determining the exact behaviour of the agents from their privileges specified in LSD. The transformed ADM program can be executed directly by an ADM interpreter. A typical output from the ADM program is a textual commentary recording the events that occur during the animation. Alternatively, the ADM program can be translated automatically into an EDEN program using a translator I have written for the purpose. The translated EDEN program can then be supplemented with other definitive scripts, such as SCOUT and DoNaLD. These additional definitions may serve as a graphical interface to the simulation. Graphical interpretation of the current definitive state, such as Figure 8.2, may, therefore, be obtained in addition to a commentary.

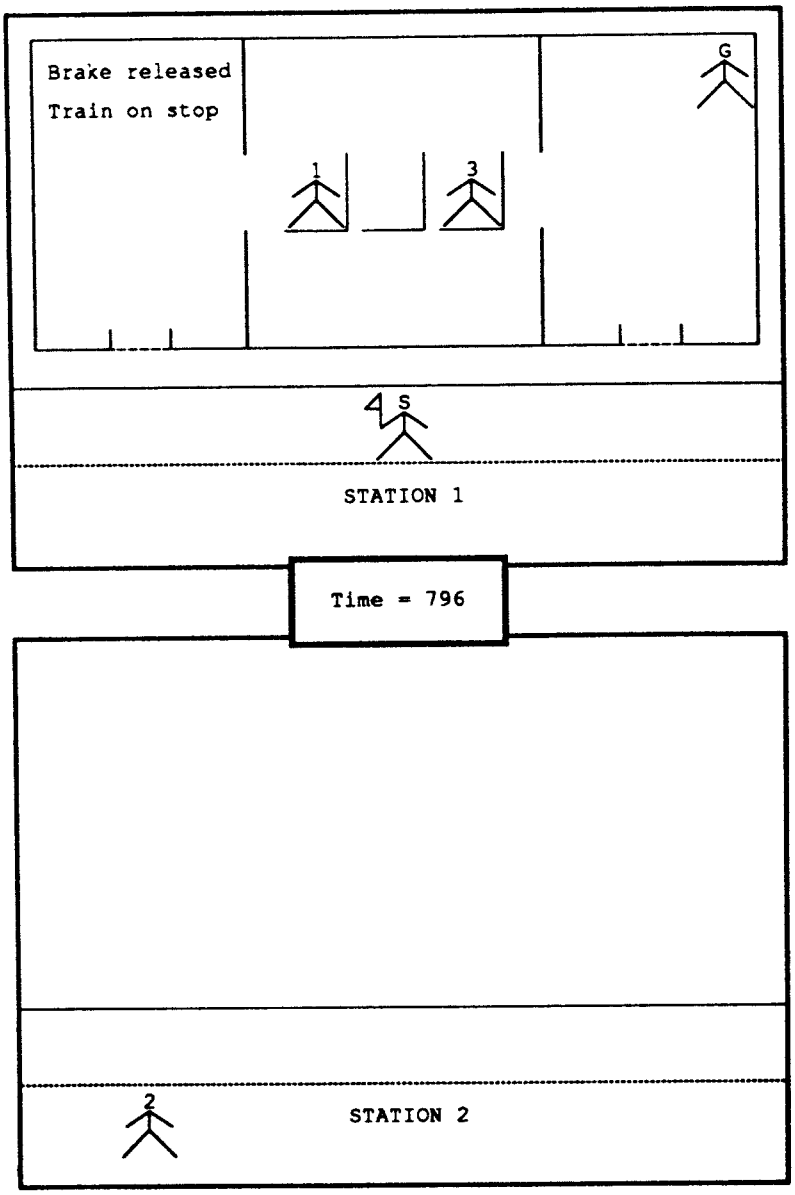

Figure 8.2: Sample Display of the Railway Station Simulation

The suggestions for enhancing the LSD notation given below, combined with the ADM-to-EDEN translator prototype will greatly reduce the time needed to produce an executable program from an LSD specification. In this way, the LSD programming environment will be better adapted for exploratory programming.

8.1. The Railway Station Simulation

A railway station simulation will be used to illustrate the process of animating an LSD specification. In this simulation we animate a train with some travellers commuting between two railway stations. The train arrival/departure protocol takes the following form:

> As the train approaches the station
>
> the guard applies the brake to stop the engine
>
> After the train has waited at the station for an appropriate interval of time
>
> the station-master checks and shuts the doors
>
> Meanwhile passengers are alighting and boarding the train
>
> When all doors are closed
>
> the station-master whistles to call the attention of driver and guard
>
> The guard waits for the station-master to raise his flag
>
> to signal the release of the brake
>
> The driver gives the ready signal to the station-master who raises his flag
>
> After the brake is released
>
> the guard signals to the station-master by raising a flag
>
> The station-master signals the driver to start the engine.

The agents involved are identified from the above protocol, and each agent is described by an LSD agent. Some agents involved in the simulation are personnel who are continuously determining their next action; they are the *station-master*, the *guard*, the *driver* and the *passengers*. Some agents are passive objects that are manipulated by other agents or are routinely doing a job; they are the *train* and the *clock*. A door should also be a passive object, but at most one person can pass through a door at a time, and in order to simplify the protocol for door use by passengers, a *door* agent in our specification includes a mechanism to choose which passenger should pass through when several attempt to pass through simultaneously.

Having identified the agents, we identify what attributes they possess (the *state* variables), what they can perceive in the environment (the *oracle* variables), what can they change in the environment (the *handle* variables), what knowledge they can derive from the things they perceived (the *derivates*), and what privileges make up their protocols. As an example, Listing 8.1 shows the LSD specification of the station-master agent. The complete LSD specification of the railway station simulation can be found in Appendix H.

```
agent sm() {                                    // The station master:
state     (time) tarrive = ⌈Time⌉;              // registers time of arrival
          (bool) can_move = false;              // determines whether the driver can start the engine
          (bool) whistle = false;               // controls the whistle
          (bool) whistled = false;              // remembers whether he has blown the whistle
          (bool) sm_flag = false;               // controls the flag
          (bool) sm_raised_flag = false;        // remembers whether he has raised the flag
oracle    (time) Limit                          // knows the time to elapse before departure due
          (time) Time;                          // knows the current time
          (bool) guard_raised_flag;             // knows whether the guard has raised his flag
          (bool) driver_ready;                  // knows the driver is ready
          (bool) around[d]; (d = 1 .. number_of_doors)
                                                // knows whether there's anybody around doorway
          (bool) door_open[d]; (d = 1 .. number_of_doors)        // the doors status
handle    (bool) can_move, whistle, whistled, sm_flag, sm_raised_flag;
          (bool) door_open[d]; (d = 1 .. number_of_doors)        // partially controls the doors
derivate  (bool) ready = ∧ (¬door_open[d]) | d = 1 .. number_of_doors);
                                                // monitors whether all doors are shut
          (bool) timeout = (Time − tarrive) > Limit;        // monitors whether departure is due
protocol  door_open[d] ^ ¬around[d] -> door_open[d] = false (d = 1 .. number_of_doors)
          ready ^ timeout ^ ¬whistled -> whistle = true; whistled = true[1]; guard(); whistle = false
          ready ^ whistled ^ ¬sm_raised_flag -> sm_flag = true; sm_raised_flag = true
          sm_flag ^ guard_raised_flag -> sm_flag = false
          ready ^ guard_raised_flag ^ driver_ready ^ engaged ^ ¬can_move -> can_move = true
}
```

Listing 8.1: An LSD Specification of a Station-Master

8.2. Terminology in the LSD Notation

LSD is a specification language for concurrent applications. Although the terminology used in LSD has been changed since the first discussions of it in the papers

---

[1] See §8.3 for the reason of underlining these definitions.

[Beynon86b, BN87], the principle underlying LSD remains unchanged. LSD describes a system by describing the behaviour of the individual agents involved in the system.

In Mike Slade's definition of LSD [Slade89], an agent description takes the following form:

```
agent agent_name (parameter_list) {
oracle list_of_oracle_variables
state list_of_state_variables
derivate list_of_derivate_variables
protocol list_of_guarded_actions
}
```

A *guarded_action* takes the form

```
boolean_condition -> list_of_actions
```

and an *action* is a *definition*, an *instantiation* or a *deletion* of an agent.

An LSD specification for an agent describes:

- the aspects of the system state to which it can respond – its *oracle* variables;
- those aspects it can conditionally change – its *state* variables;
- the circumstances under which state-changing actions can be performed – its *protocol*;
- definitions which can express the different ways in which agent actions are to be interpreted in state-transition terms according to the context – its *derivate* variables.

Slade's version of LSD has already deviated from that of [Beynon86b] and [BN87] in that the term *agent* is used instead of *process* (a term originally derived from SDL [BN87]). Both agents and processes describe strands of activity in a computation. The reason for the change is that *process* suggests a circumscribed pattern of state changes undergone whilst *agent* suggests active changes whose effect is yet to be circumscribed.

From previous experience of communicating the LSD notation to various audiences, we have found that the original terminology of LSD is also confusing in other respects:

1. *Types of variables*

There were three kinds of variable qualifiers in the original LSD:

state variable – variable that can be (re)defined by the agent;

oracle variable – variable whose value is known by the agent;

owned variable – variable owned by the agent.

Owned variables were identified by a hatch sign (#) preceding the identifiers. More than one qualifier might be connected with the same variable name.

Changes to these qualifier conventions were introduced in [BBY92]. Since the qualifier "*state*" suggests variable "determining the state" of an agent, and hence owned by it, the qualifier conventions have been changed to the following:

state variable – variable owned by the agent (was owned variable);

oracle variable – variable whose value is known by the agent (as before);

handle variable – variable that can be (re)defined by the agent (was *state* variable).

As before, the same variable name can have more than one qualifier.

2. *The term protocol*

There is a sense in which the term *protocol* suggests a rigid pattern of execution for the guarded actions. In fact, the protocol of an LSD agent should be interpreted according to the following steps:

1. All the guards are evaluated.

2. If at least one of the guards is true then a guarded action is chosen arbitrarily, otherwise the guards are re-evaluated.

3. The action list associated with the chosen guard is executed sequentially.

4. The procedure is repeated.

This interpretation scheme has a non-deterministic nature. There is no indication of the likelihood with which a particular guarded action is chosen; there is also no indication of the delays between actions in the action list. So the LSD specification specifies what the agents *can* do rather than what the agents *will* do under certain circumstance. The actual behaviour may differ according to the simulation decisions made. The term *privilege* more accurately describes a guarded action.

The term *protocol* remains meaningful in the sense that the group of guarded actions does in fact describe the privileges in an agent's protocol. For example in a railway station simulation, the LSD specification is describing a train arrival/departure protocol. Therefore the term *protocol* is still acceptable though *privilege* is arguably a better word. For the present, we have decided not to change terminology in this respect. To prevent too many versions of LSD notations being in circulation, the term *protocol* is used in this thesis. As a result, the LSD examples in this thesis, such as Listing 8.1, use the same version of LSD as in [BBY92].

8.3. Transformation from LSD to ADM

An ADM program consists of a set of entity descriptions and instances of entities. An entity consists of a set of definitive variables and a set of actions. Examples of entity descriptions are:

```
entity clock()
{
definition
      time = 0,
action
      true -> time = (|time| + 1) % (3600 * 24)
                              // 3600 * 24 = number of seconds in a day
}

entity alarm()
{
definition
      switch = true,
      alarm_time = 8 * 3600, // 8 o'clock
      alarm = switch && (time - alarm_time) % (3600 * 24) >= 0 &&
            (time - alarm_time) % (3600 * 24) < 20
                                                  // alarm on for 20 seconds
action
      alarm
         print("BEEP!")
            -> beep()
}
```

Listing 8.2: ADM Entity Descriptions of Alarm() and Clock()[2]

ADM has an unambiguous operational semantics. When an ADM program is started executing, the definitions in the definition sections of the entities are stored in the ADM's Definition Store and actions in the action sections are stored in the ADM's Action Store. Then all guards of the actions are evaluated in the context of the script associated with the definition store. If a guard is true, the message in the print statement will be displayed and the associated action(s) are stored in a Run Set. After all guards are evaluated:

- if the Run Set contains no actions, execution terminates;
- if the Run Set contains conflicting actions, for examples multiple redefinitions of the same variable, execution halts;
- otherwise, all actions are executed in parallel.

---

[2] Taken from [Slade89]

An action can be a redefinition, an entity instantiation, an entity deletion or a stop action. An entity instantiation will cause the definitions and actions of the new entity to be stored in the definition store and the action store respectively. Similarly, an entity deletion will remove the definitions and actions belonged to the entity from the definition store and the action store.

When all the actions in the Run Set have been executed, the ADM system is in a new state. The system has now completed an execution cycle. The process is then repeated.

Since the LSD notation does not specify the precise nature of interaction and synchronisation between agents, an LSD specification describes a family of possible behaviours. To execute an LSD specification, synchronisation details (we have called them *simulation decisions* or *scenario information*) must be added. Slade identified a number of issues for transforming an LSD specification to an ADM program [Slade89]. These issues concern the following questions:

(Q1) How accurate are the oracle variables? In real life, there are often some passengers who are travelling on the wrong train. This may be due to the passengers' false perception of time or platform number. In the simulation, the accuracy of the oracle variables reflects the degree of faithfulness in the model of the railway system.

(Q2) How frequently are the guards evaluated? For example: how frequently does the station-master check for time out?

(Q3) Are there any parallel actions?

(Q4) What are the response times and delays between actions?

(Q5) Are guards mutually exclusive? In other words, is an agent privileged to start more than one series of actions at the same time?

By applying the transformation techniques provided by Slade in addressing these issues, we have transformed the station-master agent in Listing 8.1 to an sm entity as in Listing 8.3.

```
entity sm() {
definition
        whistle = false,
        whistled = false,
        sm_flag = false,
        sm_raised_flag = false,
        can_move = false,
        ready = !door_open[1] && !door_open[2],
        tarrive,
        timeout = (Time - tarrive) > Limit,
        level = 0,
        init = true
action
        init
                -> tarrive = |Time|; init = false,
        door_open[1] && !around[1]
            print("Station master shuts door 1")
                -> door_open[1] = false,
        door_open[2] && !around[2]
            print("Station master shuts door 2")
                -> door_open[2] = false,
        ready && timeout && !whistled
            print("Station master whistles to call guard")
                -> whistle = true; whistled = true; guard(); level = 1,
        level == 1
            print("Station master stops whistling")
                -> whistle = false; level = 0,
        ready && whistled && !sm_raised_flag
            print("Station master raises his flag")
                -> sm_flag = true; sm_raised_flag = true,
        sm_flag && guard_raised_flag
            print("Station master lowers his flag")
                -> sm_flag = false,
        ready && guard_raised_flag && driver_ready && engaged && !can_move
            print("Train can move now")
                -> can_move = true
}
```

Listing 8.3: ADM Specification of the Station-Master Entity

This sm entity reflects the following assumptions about the behaviour of an ideal sm agent: the agent has immediate and correct knowledge of its environment (oracle variables), quickest response time[3] and minimal delay between actions in the action

[3] *Quickest response time* and *minimal delay* are with respect to the limit of ADM. That is an action will take place in the next ADM time slot when the guard becomes true; sequential actions will be performed in consecutive slots.

lists. Although these assumptions are not entirely realistic, they lead to the simplest implementation of the sm entity in programming terms.

8.4. An Evaluation of the LSD Notation

Deutsch suggests a scenario-oriented approach for programming [Deutsch89]. A scenario typically describes a stimulus-response relationship, a behaviour pattern that would be visible to a system user. Deutsch argues that this kind of description will enhance communication between non-computer experts, such as users and customers, and the software engineers who are developing the system. An LSD specification resembles a scenario-oriented specification in that a guarded action is similar to a scenario in Deutsch's sense – the guards are the stimuli and the associated actions are the responses. By Deutsch's criteria, LSD is a good specification language. In comparison, an ADM program is less concise and comprehensible than the corresponding LSD specification. For example in the Railway Station simulation example, the ADM program is about twice the length of the LSD specification and is far less readable. This is the cost of putting precise operational details into a specification.

Programming using LSD and ADM reveals a tension between intelligibility and ambiguity. Though neglecting operational details makes the operational interpretation of an LSD specification ambiguous, it also means that the specification can intelligibly describe a family of simulations. An LSD specification has to be transformed into an executable form, but we still advocate that it is a better practice to specify the program in LSD first. The operational ambiguity is relatively easily resolved by systematically addressing the simulation issues, whilst intelligibility is harder to achieve.

During the transformation to ADM, the information essential for the determination of operational behaviour is inserted. The tension between intelligibility and ambiguity means that the transformation process cannot be done automatically. It seems promising though that a hidden text annotation of an LSD specification can be

transformed automatically into an ADM program without destroying the intelligibility of the LSD notation.

The rest of this chapter evaluates the possibility of automatic translation. The following sub-sections show some suggestions for enhancing and annotating the LSD notation. When these suggestions are implemented, it is entirely possible that an annotated LSD specification can be transformed automatically into an ADM program.

8.4.1. Grouping Guarded Commands

In the present design of LSD, the guarded actions are grouped by agent in such a way that only one guarded action is chosen arbitrarily if more than one guard in the protocol is true. In general, an agent may be capable of performing two uncoordinated actions simultaneously, for example, as when a person is walking and clapping at the same time. This argues for the introduction of a hierarchical grouping of actions corresponding to a decomposition of an agent into sub-agents. In this way, more than one guarded action (at most one from each group representing a sub-agent) can be executed concurrently.

8.4.2. Parallel Action Specification

Notice that the underlined definitions in Listing 8.1 should, in principle, be executed in parallel. This is correctly reflected in the corresponding ADM entity. However, the standard interpretation of an LSD guarded action restricts the actions in the action list to be executed sequentially (cf [Slade89]). This means that the transformation of the station-master agent is not entirely faithful. On the other hand, the current LSD notation has no provision for specifying synchronised actions.

To deal with synchronised actions, a new parallel action separator could be added to LSD to specify parallel execution of actions in the action list. Associated with this change, brackets are needed to disambiguate the grouping of parallel and sequential

actions. With this parallel actions enforcing technique, we could then specify agents such as the following swapping agent:

```
agent swap() {
state     done = false
oracle    a, b, done
handle    a, b, done
protocol
          !done -> (a = |b| // b = |a|); done = true
}
```

Listing 8.4: A Swapping Agent Illustrating Parallel Actions

This cannot otherwise be specified so concisely.

8.4.2.1. Limitation of LSD for Specifying a Swapping Agent

Listing 8.5 and 8.6 are two attempts to swap the values of a and b; both attempts fail. Listing 8.5 fails because the action list associated with the chosen guard is executed sequentially: the result of execution will be that both a and b will be defined as the original value of b. Listing 8.6 fails because initially both guards are true, just one of the guarded action list is chosen arbitrarily rather than both, with the result that both a and b will acquire the original value of either a or b non-deterministically.

```
agent swap1() {
state     done = false
oracle    a, b, done
handle    a, b, done
protocol
          !done -> a = |b|; b = |a|; done = true
}
```

Listing 8.5: A Swapping Example (1st Attempt)

```
agent swap2() {
state       adone = false
            bdone = false
oracle      a, b, adone, bdone
handle      a, b, adone, bdone
protocol
            !adone -> a = |b|; adone = true
            !bdone -> b = |a|; bdone = true
}
```

Listing 8.6: A Swapping Example (2nd Attempt)

```
agent swap3() {
state   done = false, temp
oracle  a, b, temp, done
handle  a, b, temp, done
protocol
        !done -> temp = |a|; a = |b|; b = |temp|; done = true
}
```

Listing 8.7: A Swapping Example (3rd Attempt)

Listing 8.7 is the third attempt to the problem. The values of the variables a and b are successfully swapped by using a temporary variable, as in a conventional procedural program. Cognitive interpretation of the actions of the first three attempts can be made by imagining the agents are trying to move items between boxes, an item at most can be put in each box at any time. Swap3 exchanges the items inside the boxes a and b by moving one item at a time, this method requires one extra box but needs one hand only; swap1 and swap2 try to exchange the items simultaneously, it needs two hands to pick up the two items and then replace them in position at the same time. Listing 8.8 shows how this two-hand idea may be implemented in LSD. This implementation is very inefficient because it involves i) many variables, ii) instantiation and deletion of agents and iii) relatively complex synchronization between agents atob and btoa. Furthermore, this implementation still cannot guarantee simultaneous execution of actions. The fundamental weakness of the LSD agent is the inability to perform two actions in parallel by an agent. This restricts what an agent could do; it is also an undesirable feature in terms of concurrency (we would like to do as many actions in parallel as possible).

```
agent swap() {
state    done = false, start = true
oracle   start, adone, bdone
handle   start, done
protocol
      start -> atob(); btoa(); start = false
      adone && bdone -> delete atob(); delete btoa(); done = true
}

agent atob() {
state    a_val = |a|, held_a = true, bdone = false
oracle   a, a_val, held_b
handle   held_a, b, bdone
protocol
         held_b -> b = |a_val|; bdone = true
}

agent btoa() {
state    b_val = |b|, held_b = true, adone = false
oracle   b, b_val, held_a
handle   held_b, a, adone
protocol
         held_a -> a = |b_val|; adone = true
}
```

Listing 8.8: A Swapping Example (4th Attempt)

8.4.3. Call-by-Reference Parameter

Another problem with all five attempts to specify a swapping agent (Listings 8.4 through 8.8) is a lack of generality; each swapping agent can – and is intended to – swap variable a with variable b specifically. Clarification of the conventions for giving parameters to the LSD agents is required in this situation. If LSD agent parameters are to be interpreted as call-by-value parameters, the variables in the parameter list cannot be redefined[4]. Swap(a, b) does not allow redefinition of a and b, and cannot swap the two. The swapping agent in Listing 8.9 illustrates a proposed syntax for call-by-reference parameters. Like the call-by-value parameters for LSD agents that were described by Slade in [Slade89], call-by-reference parameters serve two purposes: i) to pass information to the agent to be instantiated and ii) to disambiguate identifiers of the

[4] The C language has only call-by-value parameters, but it provides the & and * operators to return the address and the content of a variable respectively; this achieves the same effect as call-by-reference.

agent instances. In respect of ii), where the *values* of the call-by-value parameters are used to identify agent instances, call-by-reference parameters make it possible to identify agent instances using parameter *names*. Call-by-reference parameters also have the advantage that they can be redefined by the instantiated agent.

```
agent swap()[a,b] {
state     done[a,b] = false
oracle    a, b, done[a,b]
handle    a, b, done[a,b]
protocol
   !done[a,b] -> (a = |b| // b = |a|); done[a,b] = true
}
```

Listing 8.9: A Swapping Agent Using Call-by-Reference Parameters

8.4.4. Hidden-Text Annotation

The above suggestions have addressed those simulation issues raised by questions (Q3) and (Q5) in §8.3. The other simulation issues could be addressed by inserting *hidden-text* into an LSD specification. By *hidden-text* I mean the use of a programming interface in which the text that accompanies an LSD specification is not normally shown or editable unless specifically requested by the programmer. For example, an interface may be designed in such a way that a double-click of the mouse on an oracle variable will open a simple script, which specifies the relationship between the agent's perception of the variable and its authentic value[5]. In a similar spirit, we may associate buttons with the guarded actions, so that the frequency of guard evaluation, action responding time and the delays between actions can be recorded and modified without actually changing the LSD specification.

This way of annotation does not alter the interpretation of LSD (an LSD specification is still describing a family of behaviours) but, at the same time, provides a

[5] The *authentic value* of a variable is the value associated with its (unique) occurrence as an owned variable.

convenient representation for the simulation decisions required in animating a particular operational behaviour.

8.5. Translation from ADM to EDEN

8.5.1. Motivation

The current ADM language has two serious limitations:

1. The only output channel of ADM is via the print statement. This method is best suited for providing information in a procedural fashion rather than describing a state in a definitive manner. In contrast, the Scout system aims at graphical representation of state. Since Scout is an EDEN-based system, translating ADM programs into EDEN programs will greatly enhance the presentation of the ADM simulation corresponding to an LSD specification.

2. ADM has a highly limited underlying algebra. The current ADM language has only the *integer* data type. This restricts the range of LSD specifications that ADM can simulate. This restriction can be overcome if an LSD specification can be simulated by a system accepting different definitive notations. This section shows that it is possible to translate from ADM to EDEN. This means that the transformation techniques described in §8.3 can be adapted in principle to simulate LSD in the Scout system directly.

8.5.2. The Translation Scheme

An entity instance in ADM comprises two parts: the *definition* part and the *action* part. A definition in ADM can be simulated as an EDEN definition; a guarded action in ADM can be simulated by an if-statement in EDEN. The main difficulty in the translation comes from the fact that ADM performs actions and redefinitions in parallel while EDEN is basically a sequential language. ADM divides the system time into slots. In the first slot the guards are evaluated and the actions to be performed are recorded in an

action store. The actions are then performed in the second time slot and the guards are re-evaluated. The actions caused by the re-evaluation of guards are performed in the third slot and so on. On the other hand, consider the following plausible EDEN implementation of two ADM guarded actions:

```
if (guard1) { action1(); }      /* guard1 -> action1() */
if (guard2) { action2(); }      /* guard2 -> action2() */
```

action1() may change the value of guard2 that result in a false invocation or suppression of action2(). This means that the evaluation of guards and performance of actions must be separated in different time slots in order to avoid interference. The solution employed in our translator is based upon delaying the execution of actions by means of saving the actions in a *message queue* (the same communication method used between EDEN and the X window interface EX).

```
proc clocking : sysClock, stopClock {
   if (!stopClock && sysClock < stopTime) {
      if (Pause > 0)
         sleep(Pause / 2);    /* delay for Pause/2 seconds */
      if (sysClock != -1) {
         SendToEden("sysClock = -1;\n");
                  /* SendToEDEN sends an EDEN statement to EDEN
                     via a message queue */
      } else {
         nextClock++;
         if (!Silent)
           SendToEden("writeln(\"time = \", nextClock);\n");
         SendToEden("sysClock = nextClock;\n");
      }
   }
}

stopClock = 1;     /* set stopClock to stop clocking */
stopTime = 30;     /* set stopTime to the system exit time */
Silent = 0;        /* set to suppress showing of time */
Pause = 1;         /* minimum gap between two system clock pulses */
```

Listing 8.10: EDEN Simulation of a Two-Phase Clock

The use of the message queue is similar to that in the simulation of a system clock in the last chapter, except that a two-phase clock is simulated here. In Phase I, all guards are evaluated and the actions to be taken are sent to EDEN via message queue; in

Phase II, all actions in the message queue are retrieved and executed. In effect, the message queue becomes a buffer similar to the action store in ADM. The two-phase clock is simulated by the EDEN action and definitions in Listing 8.10.

When stopClock is unset (defined as 0), the clocking action will start. As a result the clocking action will be continuously invoked unless stopClock is set again or the predefined stopping time, stopTime, is reached. This is because in each invocation, clocking will generate a redefinition of sysClock, which is one of the triggering variables of the clocking action itself. The redefinition will become active only after all the redefinitions and actions in the current phase are executed. If the variable Silent is 0 (default value), a message showing the current system time will be displayed. This function is not essential when the program is executed in conjunction with the Scout system, since Scout can be used directly to display time. The Pause variable sets the minimum time between two system clock pulses. Since in between two clock cycles, there may be different number of actions taking place, setting a minimum clock rate will make the simulation run more evenly (but more slowly).

Using this two-phase clock, an ADM guarded action is translated into an EDEN action in the way illustrated by the following example. A guarded action of the station-master (sm) entity is:

```
ready && whistled && !sm_raised_flag
      print("Station master raises his flag")
            -> sm_flag = true; sm_raised_flag = true
```

Its EDEN translation is:

```
proc sm_action_6[6] : sysClock {
   if (sysClock == -1) return;
   if (isTrue(ready) && isTrue(whistled) &&
         !isTrue(sm_raised_flag)) {
      writeln(\"Station master raises his flag\");
      SendToEden(\"sm_flag is TRUE; sm_raised_flag is TRUE;\\n\");
   }
}
```

[6] The name sm_action_6 corresponds to the sixth action of the sm entity.

When sysClock is –1, i.e. in Phase II, the if-statement will not be executed; otherwise the guard as in the condition of the if-statement is evaluated and the possible actions will be executed in the next Phase II (sysClock == –1).

An ADM entity is translated into a group of EDEN definitions and actions. An ADM entity specification is therefore translated to an EDEN procedure which generates the corresponding definitions and EDEN actions. The instantiation of an entity is the execution of this procedure. The deletion of an entity is the removal of the definitions and actions from the EDEN interpreter (this is done by means of the forget statement of EDEN).

The whole EDEN translation of the sm entity is too long to be included in the main text. The interested reader can see Appendix H for the whole Railway Station Simulation Example in LSD, ADM and EDEN.

The current deficiency of this translation scheme is that it cannot detect conflicts, whilst the ADM translator can. Conflict detection cannot be done easily because this EDEN implementation does not have a run set equivalent to that of the ADM interpreter. The actions in the message queue should not be treated as parallel actions because the message queue is also used for communicating between EDEN and the EX graphics interface. Without a proper run set, analysis of the parallel actions cannot be performed. A better translation scheme should therefore include a proper simulation of a run set. However, the current translator has demonstrated the possibility of automatic translation from ADM to EDEN. Taking account of the suggestions for enhancing the LSD notation in §8.4, there is then a good prospect of using LSD more conveniently and of overcoming the current limitations of ADM. In this way, the LSD programming environment will become suitable for exploratory development.