Chapter 5

Modelling and simulating concurrent systems

Adde parvum parvo magnus acervus erit

[Add little to little and there will be a big pile]

Ovid

§5 Modelling and simulating concurrent systems

Following Zeigler [1976, p.3] we observe that the process of "modelling and simulation" has three major ingredients: system, model and computer. The system comes either from the real world or is imagined, and exhibits some observable behaviour that we wish to model. The model is a representation of the system in the form of a set of instructions that can be used to generate behavioural data, which can then be interpreted in terms of the modelled system. Simulation is the process of using the model to replicate the system, typically (though not necessarily) performed by a computer. Modelling is concerned with the relationship between the system and the model, and simulation is concerned with the relationship between the model and the computer.

Two different perspectives can be identified when considering the simulation of a concurrent system: event-oriented and activity-oriented [Nance 81]. An event-oriented view focuses on describing the behaviour of the system in terms of the actions which are performed in the simulation, and an activity-oriented view describes when agents in the model can act, and what actions they can perform. As observed in [Beynon et al 88a], "*In the systematic development of sound concurrent systems software, there is in general an important need to adopt both viewpoints: to abstractly reason about the behaviour of a system and at the same time formally describe the rôles of the participating agents*".

In this chapter we describe an activity-oriented method for describing the modelling and simulation of concurrent systems, which uses a definition-based specification language called LSD (Language for Specification and Description), introduced in [Beynon 86]. It is an activity-oriented method in the sense that individual agents in the system are modelled. LSD is intended for describing systems at a high degree of abstraction by not specifying the interaction and synchronisation of agents, so an LSD specification may describe a family of behaviours.

The behaviour described by an LSD specification is explained in §5.2, with synchronisation issues discussed in §5.3. §5.4 shows how an LSD specification can be transformed into a family of programs for the adm, which allows simulation of the system which has been modelled using LSD. In §5.5 we discuss what an LSD agent represents and show how LSD can be used for modelling.

§5.1 What is LSD ?

LSD is a language for specifying and describing a communicating system of processes acting concurrently. An LSD specification consists of a set of agent descriptions. It is a process-oriented notation (as in [Andrews & Schneider 83, p.4]), in the sense that each agent is a process, and acts autonomously. A generic agent description in an LSD specification has the 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_commands
}
```

As an example, in an LSD specification of an elevator a description of an agent to open and close the door might be:

```
agent door_control() {
oracle (bool) hold = false , moving , waited
state (bool) open
derivate (bool) #can_open = ¬moving € ¬open
protocol
  ¬waited € can_open § open = true,
  hold € can_open § open = true,
  ¬hold € open € waited § open = false
}
```

Figure 5.1

Figure 5.1 can be explained informally. The door controller can refer to whether the door hold button is being pressed, whether the elevator is moving, and whether the doors have waited open for long enough. It can change the status of the door, i.e. open or close it. A necessary prerequisite for opening the door is that the elevator is neither moving nor currently open. There are three actions which the door controller can perform (as given in the protocol): it can open the door when the elevator has stopped and the doors have not already been opened for the appropriate time; it can open the doors when the hold button is pressed; it can close the doors when the doors have been open long enough and the hold button is not being pressed. We will explain agent descriptions with reference to Figure 5.1.

Agents can be instantiated. An agent instance is identified by its name and parameter list, in the same way that an entity instance is identifed by the name of the generic entity description and the parameter list. In Figure 5.1 the

agent is called `door_control` and it takes no parameters. An agent description contains a variable declaration and a protocol of guarded commands. The variable declaration describes the variables which can be referred to by the agent. The protocol describes the manner in which the agent can act to change the state.

We initially explain the interpretation of an LSD specification in purely behavioural terms, giving a precise semantics to a specification by showing how to convert it into a family of adm programs. In §5.5 we discuss what an agent actually describes. The nature of an LSD specification cannot be fully understood just in terms of a formal description of its interpretation, although this is a necessary component, but also requires reference to §5.5. This means, for instance, that it would be inappropriate to extrapolate from the formal description of an LSD specification to the mechanism for the maintenance of derivates. The reader can also refer to the examples of using LSD in Chapters 6 and 7.

In the next two sections we describe variables and protocol descriptions.

§5.1.1 Variables

Variables are of one of three *kinds*: oracle, state and derivate. An oracle of an agent is a variable whose value can be referred to, but not changed, by the agent. A state variable is a variable whose value can be both referred to and conditionally changed by the agent. A derivate is a variable whose value is defined by a formula in terms of other variables, as described in §2.1.1. In Figure 5.1, `hold` is an oracle for `door_control()`, `open` is a state variable, and `can_open` is a derivate. The type of a variable specifies the domain of values over which it may range, for example integer, boolean or string. In Figure 5.1 all variables are boolean. The declaration of a variable in an agent description specifies its kind (**`oracle`**, **`state`** or **`derivate`**) and its type (e.g. integer or boolean).

Variables in an agent instance can take parameters. Variables in the generic agent description can only use parameters which appear in the formal parameter list. Parameters have two rôles in LSD: as disambiguators and as an information passing mechanism. They can be used to allow several uses of the same agent to be distinguishable. If an agent does not take parameters, only one instance of the agent can be used at any one time (e.g. only one door controller can exist in the system). Similarly a parametrised agent cannot be simultaneously in use twice with the same parameter list.

Agents can own variables. A variable is denoted as owned by prefixing its declaration within the agent by a "#". In Figure 5.1 `can_open` is owned by `door_control()`. A variable being owned by an agent means that reference to the variable only has meaning when the agent is instantiated.

On instantiation of an agent variables may be initialised. Oracle and state variables can only be assigned explicit values or undefined. If a variable is declared as an oracle, it may only be initialised if it is owned by that agent (in the same way that, although age is an oracle for a person, when she is born her age is initialised to zero). If a variable is declared as a state, it may or may not be initialised.

Variables are used for communication between agents. A variable can be declared by more than one agent, although it can only be owned by one agent. For example, what is a state variable for one agent (e.g. `open` in Figure 5.1) may be referred to as an oracle by another (e.g. by an agent which causes the elevator to move up an down). A derivate of one agent cannot be referred to as anything other than an oracle by another. The use of the same name for a variable in different agents is interpreted as indicating that the variable will be used for communication in this way.

Informally, the use of a variable by one agent as a state or derivate and by another as an oracle allows changes in the value of the variable to be communicated from the first to the second agent. The use of common variable names to indicate communication does not, however, designate a "shared variable" in the conventional sense. Shared variables use shared memory, where two or more agents can inspect the same area of memory where the value of the variable is stored. In an LSD specification each agent has its own private copy of the value of the variable, and there is some means of communication between agents which informs an agent of the new value of a variable when it has changed. For a formal description of this method of communication refer to the transformation process described in §5.4.1.

§5.1.2 Protocols

The second part of an agent description is the protocol for action. The protocol contains a set of actions, where an

action is a guard and a command list. A guard is an enabling condition for execution of the command list. A command list is a list of commands to be executed sequentially, where each command is either an assignment to a state variable (for example in the protocol of `door_control()` in Figure 5.1 where `open` is assigned a new value) or the dynamic instantiation or deletion of an agent.

If the command is an assignment of a value to a state variable (e.g. "`open = true`") then the value of the variable which owns it will be updated accordingly. A state variable can be assigned a value or undefined (i.e. "@"). Only state variables can be directly assigned values because oracles cannot be changed by the agent, and derivates can only have their values changed indirectly by changing the values of variables on which they depend. The reassignment may involve evaluation with the evaluation operator (e.g. "`start_time = |time|`"). Parametric variables may use other variables or constants for parameters. Variables used as parameters are evaluated when the command is performed.

If the command is an agent instantiation (e.g. "`door_control()`") then the formal parameter list in the agent description is substituted by the evaluated actual parameter list in the command, and the new agent is brought into being. All parametric variables in the instantiated agent have parameters substituted by constants from the actual parameter list. The instantiated agent then starts asynchronous execution.

If the command is an agent deletion then all variables which that instance of the agent owns cease to exist and the agent instance is removed from the system. There are two ways in which an agent instance can be deleted: by the use of the **`delete`** keyword or by making the `LIVE` variable become false. Any agent instance can be explicitly deleted by the first method when the agent name and appropriate parameter list are supplied, e.g. "**`delete`** `door_control()`". The other way to delete an agent is by the use of a special variable called `LIVE`. Each agent has associated with it a private (in the sense that it is not visible to any other agent) variable `LIVE`. The `LIVE` variable may or may not be declared. If it is not declared, it is a state variable which can be assigned the value **`false`** in a command, which has the effect of deleting the agent. If the variable `LIVE` is declared then it must be as a derivate, and have an associated definition. In this case the agent is deleted when the derivate becomes false. An agent may be able to indirectly delete another agent by causing its LIVE derivate to become false.

§5.2 Behaviour described by an LSD specification

An LSD specification supplies descriptions for each agent in the system. This description is interpreted as describing the behaviour of a system of concurrent asynchronous agents. To interpret the specification in this way requires instantiation of agents to supply the initial configuration and simulation of asynchronous execution of the protocols in each agent instance.

The instantiation of agents involves supplying the name and parameters (if any) of each agent instance which is in the system when the simulation starts. An instantiation of the `door_control()` agent of Figure 5.1 would be performed by the command

```
door_control()
```

and the instantiation of a parametrised agent, such as an item in a supermarket which is parametrised by a bar code and an identification number, would be performed by the command

```
product(bar_code,ident_no)
```

Each agent which is instantiated has associated with it a list of variables and a protocol of action. In the initial configuration which occurs when all the appropriate agents have been instantiated, all variables which are referred to by agent instances should be owned by exactly one agent instance.

The protocol of an agent represents the capabilities of the agent to act within the system to change the state. The state of a simulation is given at any point for each agent by a combination of procedural and declarative elements. The oracle and state values constitute the procedural component, since they have explicit values. The derivates represent the declarative elements, in that they represent the functional abstractions being made by agents. The notion of a global, or system, state during execution is subtle, since agents will be in transition at different times.

The intended interpretation of the protocol of an LSD agent is:

- all the guards are evaluated
- if at least one is true then a guarded command is chosen arbitrarily, otherwise the guards are

re-evaluated

- the command list associated with the chosen guard is executed sequentially
- the procedure is repeated

Whenever required guards and parameters should be evaluable, which requires the avoidance of a circular definition or reference to a variables which is not owned by any instantiated agent or cannot be evaluated. Four errors can occur with dynamic actions: a dynamic action on an agent for which there is no description, instantiation of an agent which has already been instantiated with the same parameter list, deletion of an agent which is not extant, and a variable being owned by the instantiated agent which is also owned by another agent instance. Variables being reassigned must be state variables which are owned by an instantiated agent.

All agents are executed asynchronously, so there is no synchronisation between the execution of the agents in the system. Each agent executes no more than one command at a time. An LSD specification describes a concurrent simulation because the different agents in the system can operate in parallel. In the adm all guards which are true in an execution cycle have their associated command list executed: there is an obligation on the entity to execute its command list. In LSD a guard being true gives permission (cf. [Maibaum et al]) for the agent to act to change the state. There can be an arbitrary time interval between the selection of a true guard for execution and the sequential execution of its command list, which reflects the fact that a guard is an enabling condition.

§5.3 Synchronisation

In this section we consider the nature of synchronisation which is to occur when interpreting the behaviour described by an LSD specification. There are two mechanisms for synchronisation in an LSD specification: perception and assumption. We examine each in turn.

§5.3.1 Synchronisation by perception

Each agent has its own private copy of the variables it can refer to, and there is a means of communication between agents which informs an agent of the new value of a variable when it has changed. We call *authentic* the value of a

variable as stored by the agent which owns it. We call *perceived* the value of a variable as stored by agents which do not own the variable. The authentic value and any perceived values of the same variable will be stored in private memory locations in different agents.

To guarantee that the perceived value of a variable is always the same as the authentic value requires that a change in the authentic value causes an immediate change in the perceived value in other agents. There is no synchronisation of states and transitions between agents, i.e. there is no global state.

When a state variable is changed by an agent, the new value is recorded in the agent which owns the variable. When this action has been performed the state of the system has been changed. When the change is registered by other agents which have the variable as an oracle will be dictated by the manner in which the specification is interpreted as describing behaviour. This mechanism of state-oracle couplings allows communication between agents by synchronisation through perception. In Figure 5.1 the value of `open` would dictate whether an enabling condition was met in another agent which allowed the elevator to ascend or descend. The closeness of the synchronisation between perceived and authentic values will dictate how closely agents can synchronise in this manner.

Sometimes it will be necessary to ensure that there is a very close synchronisation between the change occurring and being registered. An example of this in Figure 5.1 is the need, because of safety, to ensure that the perceived value of `moving` is always as accurate as possible.

Close synchronisation can be represented in an LSD specification by using a derivate which refers to a variable which is not declared by the agent. Whenever the value of the derivate is required the authentic value of the undeclared variable will be used. This will in practice require low-level synchronisation and communication between the agent with the derivate and the agent which owns the undeclared variable.

The use of the `LIVE` variable as a derivate referring to authentic values allows reasoning about when agents are extant. As an example two mutually interfering agents cannot both be extant at the same time if the conjunction of the definition of two `LIVE` derivate variables evaluates to false, provided the appropriate synchronisation by perception occurs.

In other applications it will not be necessary to maintain a close synchronisation between authentic and perceived values. In Figure 5.1 it is not necessary that the `door_control()` agent register immediately when `waited` becomes false, but it must register such a change within a reasonable time. Globally maintained relations between elements of the system are expressed by the use of derivates and state-oracle couplings.

§5.3.2 Synchronisation by assumption

The second mechanism for synchronisation between agents is through independent assumptions about the relative speeds of operation of the different agents in the system. For example, in Figure 5.1 suppose it is desired that the elevator door be opened whenever the hold button is pushed and the lift is stationary with its doors shut. It is insufficient to ensure that there is a close synchronisation between the authentic and perceived values of `hold`. If the perceived value of `hold` is changed quickly enough from false to true and then back to false, then the second guard in the protocol may not be evaluated whilst `hold` is true. In this case the door would (incorrectly) not be opened.

There are two ways to ensure synchronisation in the above example. The first is to introduce new variables to allow synchronisation by perception. An example of this might be a variable `has_pressed` which is set to true whenever the hold button is pushed, and reset when the `door_control()` agent has registered the press. This is not a satisfactory method because it would be an extremely cumbersome mechanism for any practical application and it does not represent well the way that such systems synchronise in real life.

The second mechanism to ensure synchronisation is to make assumptions about the relative operation speeds of the agents in the system. This has the advantage of conforming to the way genuine systems synchronise. For instance, it is assumed that in practice a person physically cannot press a hold button faster than can be detected by the door control mechanism. This form of synchronisation is considered in the context of a remote control system in my joint paper with Beynon and Norris [Beynon et al 88a].

In discussing (in §5.2) the behaviour described by an LSD specification, all that was said about relative speeds of execution of agents is that they operate asynchronously. In general they will operate at very different rates, a fact that will be used in the construction of the specification. For instance, the agent description of Figure 5.1 has used the assumption that the value of `hold` cannot be changed too quickly to be detected. Part of interpreting the specification as describing the behaviour of a system will involve deciding how fast agents operate relative to each

other.

§5.4 LSD specifications simulated on the adm

In this section we motivate and demonstrate the simulation of an LSD specification on the abstract definitive machine. The LSD specification can be transformed into a family of definitive programs for the adm by mechanical methods. The transformation procedure produces a family of definitive programs because different decisions can be made about the nature of the communication between agents, and the relative speeds of operation of each agent instance.

The state of a simulation can be seen as a synthesis of the states of the individual agents. A system state derived in this manner will in general contain inconsistencies, because authentic and perceived variables may differ in value. Some of these inconsistencies will be acceptable, for example certain state and oracle pairs can be allowed to differ in value. Other inconsistencies will be incorrect, for instance when `moving` in Figure 5.1 has a perceived value which differs from the authentic value. Simulation allows the investigation of what inconsistencies can be tolerated (i.e. still allow acceptable behaviour) and when perceived values must be maintained as the same as the authentic value.

We must also ensure that agents operate at the appropriate relative speeds. This constraint entails considering the maximum or minimum times between the evaluations of a guard, the time it takes to execute a command, and so on. For instance, it is necessary to ensure that the `door_control()` agent operates sufficiently quicker than the agent which presses the hold button that no press can be unnoticed.

In §5.4.1 we show how to convert an agent in an LSD specification into an entity description for the adm. This allows the mechanical transformation of an LSD specification into an executable definitive program. Techniques are outlined which allow the translation to specify relative execution times for different agents, to change how often perceived values are updated to authentic values, and to influence the choosing of guards for testing and commands for executing. Changing these parameters of the translation process allows a family of definitive programs for the adm to be derived from the same LSD specification.

The modelling and simulation of a concurrent system using LSD is performed in two stages. The language LSD is used to describe the behaviour of the concurrent agents in the system. This allows effort to be focussed on correctly describing the manner of operation of individual agents, i.e. on the abstract modelling of the system without concern for the problems caused by concurrent execution. The specification is then transformed into an executable definitive program for the abstract definitive machine, which can be used to simulate the system described by the LSD specification. The transformation process requires parameters to be supplied which describe how the agents interact. This is the most appropriate stage for consideration of interaction issues, since it is now known precisely what the agent can refer to in the state and how it can act to change the state.

It has been shown how a definitive program for the abstract definitive machine can be viewed as defining a state-transition system, and how interference can be detected when executing a definitive program on the abstract definitive machine. The simulation executing on the adm will exhibit behaviour which can be interpreted with reference to the LSD specification. If interference is detected in the transformed program then either the original LSD specification can be changed to amend the variables which can be referred to by agents or the ability of agents to change the state, or the parameters for the transformation process can be changed to allow different interactions between agents in the system. Either form of change will result in a different definitive program.

§5.4.1 The transformation

We now show how to transform an LSD specification into a family of definitive programs. The construction of the simulation entails decisions as to the relative speed of operation and how closely linked the authentic and perceived value of a variable is for each agent in the system. We use Figure 5.1 to demonstrate the early stages of the transformation, and then introduce a more complicated agent in Figure 5.2 to show how to transform agents with command lists of more than one command.

Each LSD agent is transformed into an entity description. The agent name and parameter list are used as the entity name and parameter list. A variable that is owned by the agent is put into the definition section of the entity description, together with its associated initialisation, if any. Any variable which is not owned but is initialised by the agent must be initialised by the entity. This is done by adding a new boolean variable `init_flag`, initialised to true, to the definition section. We then create a guard of the form:

`init_flag § `*`initialisation_list`*` ; init_flag=false`

where *initialisation_list* is the list of variables which are not owned by the agent but are initialised. The agent in Figure 5.1 would be transformed into:

```
entity door_control() {
definition
   can_open = ¬moving € ¬open,
   init_flag = true
action
   init_flag § hold = false ; init_flag = false
}
```

This has the effect of initialising unowned variables on the first execution cycle after the entity has been invoked, after which the guard is always false. This is not the same as initialising variables when an agent is instantiated, but is acceptable: when instantiating entities to reach the initial configuration of the system these definitions (of derivates) and assignments (of states and oracles) can be performed explicitly; when executing a dynamic instantiation it will mean that the variable is updated one execution cycle later, but we will shortly introduce a mechanism which interleaves computation described by the specification with computation to update variables. The initialisations caused by dynamic instantiations will always occur in the next execution cycle after the entity was instantiated, i.e. before any further computation from the specification occurs.

Having transformed the declared variables section of an agent we now consider the protocol. Recalling that the adm model of computation executes all commands in an entity with true guards, we must ensure that no more than one action is being executed from the set of actions in the protocol at any time. By the use of the `rand()` function we ensure that precisely one of the guards in the agent protocol is evaluated during each execution cycle. This results in Figure 5.1 being transformed into:

```
entity door_control() {
definition
   can_open = ¬moving € ¬open,
   init_flag = true,
   only_one = |rand(3)|
action
   init_flag § hold = false ; init_flag = false,
   ¬waited € can_open € (only_one == 1) § open = true,
   hold € can_open € (only_one == 2) § open = true,
   ¬hold € open € waited € (only_one == 3) § open = false,
   true § only_one = |rand(3)|
}
```

We have used the function, `rand(n)`, which returns a random integer between 1 and `n`. Choosing a larger number for n than is necessary (in this case larger than 3) allows the modelling of a door controller which does nothing in some execution cycles.

This transformation may result in some execution cycles which do not perform any commands, if the guard selected for evaluation is false. We can introduce variations on this technique for selecting guards for evaluation. We can describe the ratio between selection of the various guards for evaluation by using a range for `only_one`:

```
¬waited € can_open € (only_one ≤ 4) § open = true,
hold € can_open € (only_one ≥ 5) € (only_one ≤ 6) § open = true,
¬hold € open € waited € (only_one == 7) § open = false,
true § only_one = |rand(7)|
```

If it was required that any press of the hold button be detected, then the last guard of the entity would be:

```
true § only_one = if (hold € can_open) then 2 else |rand(3)|
```

Similarly, if it was required that the first guard of Figure 5.1 was evaluated twice as often as the others, then the last guard of the entity would be

```
true § only_one = if (rand(4) == 1) then 1 else |rand(3)|
```

which would result in only_one having the value 1 with probability 0.5, 2 with probability 0.25, and 3 with probability 0.25. This technique supplies as much control over agent execution as is required.

To simulate the asynchronous nature of agent execution it is necessary to introduce arbitrary delays between selection of a guard and the execution of the first command, and between execution of successive commands in the command list. We cannot use Figure 5.1 to illustrate this because all command lists consist of one command, so we introduce an extended version in Figure 5.2:

```
agent door_control() {
oracle (bool) hold = false , moving , waited
state (bool) open , #open_light = false,
derivate (bool) #can_open = ¬moving € ¬open
protocol
    ¬waited € can_open § open = true ; open_light = true,
    hold € can_open § open = true ; open_light = true,
    ¬hold € open € waited § open = false ; open_light = false
}
```

Figure 5.2

This agent turns a light on when the door is open. We use Figure 5.2 to demonstrate how to use delays to simulate the asynchronous nature of an LSD specification. The first guard of the protocol of Figure 5.2 is transformed into:

```
¬waited € can_open € (level == 0) € (selected == 0)
                                    § level = 1 ; selected = 1
(level == 1) € (rand(prob) == 1) € (selected == 1)
                                    § open = true ; level = 2,
(level == 2) € (rand(prob) == 1) € (selected == 1)
                                    § open_light = true ; level = 0 ;
                                      selected = 0
```

The variable `level` is used to record how much of the command list has been executed, the variable `selected` to record which action is being executed, and the variable `prob` indicates the probability that the next command will be performed in the coming execution cycle. If `prob` is 3, then there is a `0.33` chance that a command will be done in the next execution cycle. Much more precision is possible, and different probabilities can be used for different commands in the same command list. Clearly there is sufficient expressive power to represent any form of asynchronous behaviour.

The technique is scaled up for more than one guarded command. We apply this transformation to all agents in the system , so we must avoid naming conflicts between the control variables (e.g. `level`) in the different entities. We do this by the use of parametric variables, where the parameter `p` is different for each entity. The translation of Figure 5.2 into an entity gives:

```
entity door_control() {
definition
   can_open = ¬moving € ¬open, open_light = false,
   init_flag[p] = true, only_one[p] = |rand(3)|,
   prob[p] = 4, level[p] = 0, selected[p] = 0
```

```
action
init_flag[p]   § hold = false ; init_flag[p] = false,

¬waited € can_open € (only_one[p] == 1)
                                  € (level[p] == 0) € (selected[p] == 0)
               § level[p] = 1 ; selected[p] = 1,
(level[p] == 1) € (rand(prob[p]) == 1) € (selected[p] == 1)
               § open = true ; level[p] = 2,

(level[p] == 2) € (rand(prob[p]) == 1) € (selected[p] == 1)
               § open_light = true ; level[p] = 0 ; selected[p] = 0,

hold € can_open € (only_one[p] == 2)
                                  € (level[p] == 0) € (selected[p] == 0)
               § level[p] = 1 ; selected[p] = 0,
(level[p] == 1) € (rand(prob[p]) == 1) € (selected[p] == 2)
               § open = true ; level[p] = 2,
(level[p] == 2) € (rand(prob[p]) == 1) € (selected[p] == 2)
               § open_light = true ; level[p] = 0 ; selected[p] = 0,

¬hold € open € waited € (only_one[p] == 3)
                                  € (level[p] == 0) € (selected[p] == 0)
               § level[p] = 1 ; selected[p] = 0,
(level[p] == 1) € (rand(prob[p]) == 1) € (selected[p] == 3)
               § open = false ; level[p] = 2,
(level[p] == 2) € (rand(prob[p]) == 1) € (selected[p] == 3)
               § open_light = false ; level[p] = 0 ; selected[p] = 0,

selected[p] = 0 § only_one[p] = |rand(3)|
}
```

It remains to describe how we allow delays between the updating of the authentic value of a variable and the change being registered by other entities in the simulation. All variables which appear in the variable declaration section of the agent but are not owned by the agent are added to the definition section of the entity description in a parametrised form, to distinguish them from the variable in the definition section of the owning entity. We refer to these variables as perceived variables. The authentic value is stored in the authentic variable (owned by some other entity), and the perceived variable is used to store the value perceived by the instance. A parametric variable is transformed into one with the parameter `p` appended, so that `onhook[phone]` becomes `onhook[phone,p]`. Perceived variables are always initialised. If there is an initialisation of the variable given in the agent description then the evaluation of this expression is used. If no initialisation is given then the evaluation of the authentic variable is used. Figure 5.2 results in a definition section consisting of:

```
definition
   open_light = false, can_open = ¬moving[p] € ¬open[p],
```

```
moving[p] = |moving|, waited[p] = |waited|,
open[p] = |open|, hold[p] = false,
average = 3
```

The variable `average` represents the average number of execution cycles between updates of perceived variables. We use the same value for each variable, although there is no reason not to allow individual guarded commands in the action section of the entity, as done in the telephone example in Chapter 6. Guarded commands are used to periodically update the value of the perceived variable to make it the same as the authentic value. We must avoid the evaluation of a variable which is changing, so we interleave command execution and variable updating, using the technique described in Chapter 4. On every even execution cycle we perform the commands from the agent and on every odd execution cycle we update a subset of perceived variables. For Figure 5.2 this involves using the following guarded commands:

```
¬even € rand(average) == 1 § moving[p] = |moving|,
¬even € rand(average) == 1 § waited[p] = |waited|,
¬even € rand(average) == 1 § open[p] = |open|,
¬even € rand(average) == 1 § hold[p] = |hold|,
```

There will be a guarded command in the definitive program of the form:

```
true § even = ¬|even|
```

We must ensure that this guarded command occurs exactly once in the action store throughout execution of the definitive program. We do this either by placing it in an entity description which is instantiated precisely once before execution, and cannot have any dynamic actions performed on it (which can be discerned by examination of P), or by creating a new entity description which is instantiated once, and consists solely of this guarded command. The command serves two purposes: it ensures the interleaving of variable updates and actions in agent protocols, and it also ensures that there is always a true guard. If there were an execution cycle in which no guard were true, then the adm would suspend computation and await an action by the user.

We must ensure that evaluation of a variable in an agent uses the perceived value in the entity, whereas assignment to it in the agent causes a change in the authentic value. The guards of Figure 5.2 become:

```
even € ¬waited[p] € can_open
                          § open = true ; open_light = true,
even € hold[p] € can_open § open = true ; open_light = true,
even € ¬hold[p] € open[p] € waited[p]
                          § open = false ; open_light = false,
```

If the entity owns the variable no transformation is applied, since the perceived value is always the same as the authentic value. If the entity does not own the variable, then the authentic value is used on the left hand side of assignments in commands, and the perceived value used everywhere else, i.e. as parameters to instantiate entities, in guards, and on the right hand side of a reassignment. Figure 5.2 is transformed into the adm entity:

```
entity door_control() {
definition
   open_light = false, can_open = ¬moving[p] € ¬open[p],
   moving[p] = |moving|, waited[p] = |waited|,
   open[p] = |open|, hold[p] = |hold|,
   init_flag[p] = true, only_one[p] = |rand(3)|,
   prob[p] = 4, level[p] = 0, average [p] = 3,
   even[p] = true, selected[p] = 0
action
init_flag[p] § hold = false ; init_flag[p] = false,
even € ¬waited[p] € can_open € (only_one[p] == 1)
                              € (level[p] == 0) € (selected[p] == 0)
              § level[p] = 1 ; selected[p] = 1,
even € (level[p] == 1) € (rand(prob[p]) == 1) € (selected[p] == 1)
              § open = true ; level[p] = 2,
even € (level[p] == 2) € (rand(prob[p]) == 1) € (selected[p] == 1)
              § open_light = true ; level[p] = 0 ; selected[p] = 0,

even € hold[p] € can_open € (only_one[p] == 2)
                              € (level[p] == 0) € (selected[p] == 0)
              § level[p] = 1 ; selected[p] = 2,
even € (level[p] == 1) € (rand(prob[p]) == 1) € (selected[p] == 2)
              § open = true ; level[p] = 2,
even € (level[p] == 2) € (rand(prob[p]) == 1) € (selected[p] == 2)
              § open_light = true ; level[p] = 0 ; selected[p] = 0,
even € ¬hold[p] € open[p] € waited[p] € (only_one[p] == 3)
                              € (level[p] == 0) € (selected[p] == 0)
              § level[p] = 1 ; selected[p] = 3,
even € (level[p] == 1) € (rand(prob[p]) == 1) € (selected[p] == 3)
              § open = false ; level[p] = 2,
even € (level[p] == 2) € (rand(prob[p]) == 1) € (selected[p] == 3)
              § open_light = false ; level[p] = 0 ; selected[p] = 0,

even € selected[p] = 0 § only_one[p] = |rand(3)|,
¬even € rand(average [p]) == 1 § moving[p] = |moving|,
¬even € rand(average [p]) == 1 § waited[p] = |waited|,
¬even € rand(average [p]) == 1 § open[p] = |open|,
¬even € rand(average [p]) == 1 § hold[p] = |hold|,
true § even = ¬|even|
}
```

Figure 5.3

The derivate `can_open` is defined in terms of perceived variables, rather than authentic variables. It will sometimes be required that its definition is in terms of authentic variables (e.g. when it uses undeclared variables), to permit a very close synchronisation between the perceived and authentic value.

The only element of an agent that has not been transformed is the `LIVE` variable . It can be used in an agent as either a state or a derivate variable. If it is used as a state variable and there is an assignment "`LIVE = false`" in a command then this is transformed into the command "**delete** *entity_name(param_list)*". If the `LIVE` variable is used as a derivate, then its associated definition *expression* is used as the guard in a new guarded command in the action section:

¬*expression* § **delete** *entity_name(param_list)*

To avoid conflict it is necessary to avoid executing this command whenever any other computation in the entity is occurring. To allow this we change the guard to

¬even € ¬*expression* § **delete** *entity_name(param_list)*

and append a conjunct " € *expression*" to each guard of a command which assigns the authentic value to a perceived variable. The only guard which can then be true at the same time as the guard which cause the entity deletion is the one which causes initialisations of unowned variables. If the variable is to be immediately terminated on instantiation then we still want the variables to be initialised, so we change the entity deletion guarded command into:

¬even € ¬*expression* € init_flag
§ **delete** *entity_name(param_list)*

This will mean that the entity can only be deleted the second time that even becomes false, which is acceptable. Suppose an entity is to be deleted as soon as it is instantiated, i.e. its `LIVE` variable is initially false**.** The entity must be instantiated when `even` is true, since that is a prerequisite for performing a dynamic action from the specification. In the next execution cycle `even` will be false and all unowned variables initialised. Next `even` will become true, and a guarded command from the specification may be selected for execution. In the next cycle,

however, `even` will become false again. Since init_flag is now also false the entity deletion command will be true, and the entity deleted. This will occur before any commands which come from the specification have been executed.

If an LSD variable has the same name as an adm keyword, i.e. **`entity`**, **`definition`** or **`action`**, then it is changed throughout the adm program into some other variable name. Output from the execution of the adm program is facilitated by the use of procedural actions. These must be explicitly supplied for the transformation process to use.

The techniques we have described can be used for the transformation of an LSD specification into a definitive program. We have used a door control agent as an example in the description of the transformation techniques. Appendix 6 contains examples of the execution of two door control entities which were transformed from Figure 5.2 using different transformational parameters to produce different programs. Appendix 6.2 shows the results of executing the program in Figure 5.3, and Appendix 6.4 shows the results of executing a program which uses a closer synchronisation between the authentic and perceived value of the `open` variable.

§5.4.2 More elaborate transformation techniques

Sometimes more elaborate transformational techniques are required. These are briefly described here, and illustrated in the complete transformation of the telephone example in chapter 6.

When an entity can be instantiated more than once it is necessary to ensure that each perceived variable it owns is distinct from that owned by other instantiations. It is known that the parameter list for each instantiation is different, so perceived variables in the entity description are parametrised by a list of parameters consisting of the complete parameter list for the agent and the parameter `p` which is used to distinguish perceived from authentic variables. In the telephone example of Chapter 6 `onhook[_S]` in agent `connect(_S,_D)` could be transformed into the perceived variable `onhook[_S,_D,_S,4]` in the entity description. In the actual program `onhook[_S,_S,4]` is used because this is sufficient to distinguish instances of the variable: `connect(_S,_D)` never has two instantiations extant with identical first parameters.

It will sometimes be necessary when deleting an agent to reset the state to what is was before the instantiation of an entity, because the definitions which the entity has introduced are no longer appropriate when the entity is no

longer extant. This allows the problem of changing the context for action, which was described in §4.5, to be addressed. The semantics of an LSD specification in this respect has not been fixed: there are times when it is appropriate to revert to the definitions for variables which were in use before the instantiation of a transient agent, and other times when part of the function of the agent is to change the definitions of variables. The transformation of the LSD specification allows original definitions to be re-established when the entity is deleted, as in the `connect()` agent in the telephone example. This could be done mechanically by using local control variables in a transient entity to store the state when it is instantiated, and then restoring the state just before the entity is deleted.

Sometimes an entity must be slowed down, to enable the ramifications of an action it has just performed to filter through the system. An example of this is found in the telephone example of chapter 6, where the user is stopped from performing any actions for a number of execution cycles after they have dialled a number on their telephone. The reason for this is that the tone heard on the telephone does not immediately change, because of delays in propagating changes in variables and in performing actions, and so if the user were not slowed down they could immediately dial again. With the mechanism for slowing down the user which is used they can only act again once the effects of dialling the number have occurred. This mechanism uses a control variable `slow[p]`, initialised to 0, and adds a guarded command to the entity:

```
even € slow[p] > 0 § slow[p] = |slow[p]| - 1
```

The conjunct "`€ (slow[p] == 0)`" is appended to all guarded commands which come from the specification (i.e. all those which start "`even €`"), which means that they will not be executed until `slow[p]` again has the value 0. To delay further execution after a command has been executed the variable `slow[p]` is set to some value, for instance 12. This disables execution for the next 12 times that `even` is true.

Finally it is unclear what variables should be used on the right hand side of redefinitions in initialisations. Consider the following LSD specification:

```
agent one() {
oracle #a
  •
  •
  •
}
agent two() {
  •
state a = b + c
```

```
  •
  •
  •
}
```

The transformation of the two agents would result in:

```
entity one() {
definition a,
  •
  •
  •
}

entity two() {
definition
  a[1] = b[1] + c[1], init_flag[1] = true,
  b[1] , c[1] ,
    •
    •
action
  init_flag[1] § a = expression ; init_flag[1] = false,
    •
    •
}
```

It is unclear what *expression* should be. It cannot be "`b[1] + c[1]`", since if entity `two()` was deleted then that would leave `a` defined in terms of nonexistent variables. If entity `one()` owns `b` and `c`, then expression should be "`b + c`". If it does not own but perceives these variables, then perhaps the definition should use the values for `b` and `c` as perceived by entity `one()`? Alternatively (as done in the telephone example) the authentic values can be used, although this can introduce unintended synchronisations. At present these issues are resolved on a case-by-case basis, which means that this decision is another parameter for the transformation process.

§5.5 What does an agent describe ?

Pylyshyn [1986, p.3] argues that cognitive terms such as perceiving and knowing are necessary for genuinely describing or explaining some actions. He gives a fictitious example involving a car accident where an observer runs to a telephone kiosk and dials the numbers 9 and 1 (the initial numbers of the American emergency services number). In attempting to answer questions such as "What has happened ?" and "What will the pedestrian do next ?" Pylyshyn maintains that cognitive terms such as "perceive" and "infer" need to be used, i.e. that a means of describing and modelling inviolable physical laws, whilst necessary, is insufficient to genuinely explain behaviour.

The interpreting of an LSD specification involves consideration of these cognitive issues.

An LSD specification describes autonomous agents, each having some "perception" of the current state. The perception of an agent is based on what variables in the state are visible to it. Oracles and derivates are those characteristics of the system which can be observed but cannot be directly changed. State variables are those which are conditionally under the control of the agent, and can be changed by reassignments in the protocol. Guarded commands are used in an adm program either to model the consequences of physical laws (e.g. if a block in the example of Chapter 4 is pushed left, then its position changes) or to represent the "intelligent" intent of the agent (e.g. the door controller reacting to the hold button being pushed by opening the door). This point is elaborated in Chapter 7 by consideration of different types of pupil using an educational application.

Pylyshyn also notes [1986, p.85] that "*people typically have focussed on what are sometimes called "permissive" (or, in computer science, "nondeterministic") rules - for example the rules of logic or of grammar - which specify the permissible relations among representations rather than on the issue of the conditions under which such rules are invoked*". We intend that an LSD agent can be considered as having the ability to participate in such a way as to change the state, with its capabilities represented by enabling conditions on commands. This means that LSD can be described from a different, more cognitively-oriented perspective than is used in the more usual computational description, which should allow the behaviour of an agent to be more naturally described. This direct modelling of agents is a reason why parameters are required to serve as disambiguators: in the real world, all "agents" of a system are distinguishable, at least either temporally or spatially, so we wish to avoid two indistinguishable agents.

A variable is owned by an agent if it represents a property or characteristic associated with the agent, e.g. it only makes sense to consider the value of `can_open` in Figure 5.1 if the `door_control()` agent by which it is owned is in existence. The variable `open` in Figure 5.1 is not owned by the `door_control()` agent because it would be meaningful to consider whether the door was open or not even if there were no agent to open and close it - it is a property intrinsic to the door, not the door controller.

The authentic value of a variable, the value stored by the owning agent, represents the genuine value of the variable at any point in the execution. This may differ from the perceived value of the variable stored in another agent, as illustrated by the telephone example in Chapter 6. The cognitive reason for this is that real-world systems

often do not exhibit close synchronisation. There will in general be some delay between a change and it being noticed. Any instance where, unbeknown to one agent, a property of another agent has changed is an example of a lack of external consistency. Examples include having a haircut and secretly learning to play the piano. We can model this behaviour to the required degree by the appropriate use of parameters for the transformation process.