Chapter 8

Appraisal and comparison

I can call spirits from the vasty deep.

Why so can I, or so can any man; but will they come when you do call for them ?

Shakespeare, King Henry IV, Part I

§8 Appraisal and comparison

In this chapter we make an appraisal of what has been achieved by the introduction of the adm and LSD, compare definitive programming with other approaches, and indicate potential extensions for the adm and our method for modelling and simulating concurrent systems.

We describe how definitions can be used in the adm, and consider the applicability of definitive programming to interaction, the use of invariants, and parallel programming. We briefly compare definitive programming with procedural and nonprocedural programming, and consider extensions to the adm.

Definitions in an LSD specification are discussed, and an outline given of issues in LSD which have not fully been resolved. We make an appraisal of the transformation method with indications of how it can be improved, including discussion of specification simulation, analysis of the results of simulation, interaction with the simulation, and animation of the simulation. We conclude by showing how the transformation process could be the basis for a menu-driven modelling and simulation tool.

§8.1 Use of definitions in the adm

In this section we consider the use of definitions in an adm program. Two uses can be identified. A definition `housing_cost = rent + rates + bills` could be found in an adm program. If `housing_cost` was used in a guard, then the value of (`rent + rates + bills`) would be evaluated in the current state. This can be compared with the use of macros. A macro is a portion of text that is replaced by another portion of text prior to compilation. The programmer specifies the macro form and its body (the sequence of text which is to replace it), and then a macro-processor performs the appropriate substitutions. The `housing_cost` variable in a guard would act like a macro.

If all variables defined by a formula were static then whenever reference was made to a variable the reference could be substituted by the associated definition at compile time. Variables can, however, be changed dynamically in a definitive program. For instance, the formula associated with `housing_cost` might change from the above to become `rent + community_charge + bills`. Such a change could not be performed using macros,

because a macro is substituted at compile-time, before any processing has occurred. Macros do not occur in a program by the time computation begins. Only when a variable in an adm program is defined by the same formula throughout computation is it the same as a macro.

The other use of definitions in an adm program occurs in commands, when a variable is redefined. Note that if a variable x occurs in a formula used to redefine another variable y then the formula associated with x at that point is irrelevant (other than for checking that the definition of y is not circular), since no evaluation is required. The definition of x may change before y is evaluated, and since evaluation of a variable always occurs in the current state only the current definition of x is used.

§8.2 Appraisal of definitive programming

In this section we discuss the applicability of definitive programming to interaction, invariant maintenance, and parallel programming.

§8.2.1 Interaction

Gries asserts "*Programming is a goal-oriented activity*" [1981, p.173]. This is especially true of interactive systems: when solving a design problem, for instance, humans typically have a goal state in mind in which some relationships are satisfied. Transitions are performed so as to attain the goal state by an iterative process of refinement. The use of a definitive notation allows the speculative changing of variable definitions to see what effect that has on the state of the system as a whole. Definitive notations are particularly well suited to allowing an iterative approach to problem solving (e.g. [Beynon 85]), since they automatically maintain relationships whilst permitting these speculative changes to the state.

The use of definitive notations has been explored within various application areas involving interaction, including CAD [Beynon 87], simulation [Beynon et al 88a], spreadsheets and educational software [Beynon et al 89a]. The underlying algebra chosen is dictated by the application area. The automatic maintenance of relations between variables has been found to be of great utility in interactive systems.

A traditional problem with interactive systems is how to present a clear picture (known as a *snapshot*) of the current state. In a procedural program, a snapshot of the current state consists of a dump of the value of all variables. A snapshot of the current state of a definitive program would consist of the definitions of all variables. Some variables will be defined by constants of the underlying algebra, and others will be functionally defined.

The state can also be deduced from the transitions which are available. For example in the `jugs` program the `pupil()` entity cannot press a button unless the associated option is available. Were the program to be used interactively with a graphical interface then the "`Fill A`" option would not be available if jug A was full. The state of the system can thus be apprehended by observing what transitions are available.

A definitive program for the adm consists of a description of transitions between definitive states, which removes the necessity for interaction with a user. It may be desirable, however, to construct a definitive program with some variables left undefined, or defined in a non-evaluable manner. As long as the value of these variables is not required, computation within the adm can continue. In principle this allows software prototyping, the development of a preliminary version of a system to permit certain aspects of its behaviour to be studied without the need to construct the entire system. Prototyping is particularly useful for the rapid generation of a skeletal version of the final system, which a prospective user can judge as to its relevance and usefulness for the intended application. Definitive programming can naturally accommodate unspecified values as being undefined.

The feature of using the same method to interact with and program the adm is not unique to definitive programming. Having described embedded SQL, a database programming language, Date observes "*The use of essentially the same language for both interactive and programmed access to the database has one very significant consequence: it means that the database portions of an application program can be tested and debugged interactively...the interactive interface provides a very convenient programmer debugging facility*" [1985, p.200]. The same is true in an adm program. For instance, when a variable is undefined and its value is required, the program does not crash, but instead awaits input. This allows easy debugging of the program because the state in which the error has occurred is immediately accessible for interrogation. This contrasts markedly with the necessity to scatter print statements throughout a procedural program when debugging it, both to obtain a description of the state and to discover where in the program the error is occurring.

In a similar way Chmilar and Wyvill suggest that their data structure for integrating modelling and animation means that "*by specifying animation data directly in the models and structure of a scene, we can carefully avoid redoing calculations if their results have not changed since the last frame.*" [1989, p.260]. The judicious use of definitions can also allow the avoidance of unnecessary evaluations. They also observe that their approach "*encourages the animator to think of the scene as unfixed, changing over time; time becomes a factor in building a model: it looks different at different times*" [1989, p.260]. We also envisage that some of the underlying relationships in the application will evolve over time, specifically with user intervention. The distinction that is commonly made between design and execution of a program is diminished in a definitive program, since the user can interactively re-design (i.e. redefine) an underlying relationship.

§8.2.2 Invariants

To motivate the discussion of invariants, we describe their use in verification research. Work has been done [Gries 81] [Dijkstra & Feijen 88] on attempting to derive programs from specifications. The thesis underlying this approach is that programs can be derived from the postcondition by the correct application of mathematical rules, expressed as algebraic laws [Hoare 86]. Such application requires both mathematical aptitude and mechanical computation, but leads to a verified program. In other words, computer programs can be proved correct. This is because "*Computer programming is an exact science, in that all the properties of a program and all the consequences of executing it can, in principle, be found out from the text of the program itself by means of purely deductive reasoning*" [Hoare 69, p.19][1].

In Dijkstra's weakest precondition method [1976] a specification takes the form of a postcondition, which characterises the desired final state of the program execution. This is then transformed, by the intelligent application of mechanical rules, to derive a verified piece of code whose execution results in the desired postcondition. Predicate transformers are used to derive the weakest precondition such that execution of a given instruction will terminate

[1] It has been argued [DeMillo et al, p.271] that social processes play a part in our acceptance of the validity of a proof. Since program verifiers cannot take part in any such interaction, they are predestined to fail in their purpose of demonstrating the correctness of a program. Another objection focuses on the nature of causal systems, and the need to verify the verification of such a system [Fetzer 88], and draws the same conclusion. The essential nature of these arguments is that deductive reasoning is insufficient to characterise the behaviour of computer programs, a claim that is strongly refuted by formal methods proponents. We do not concern ourselves here with the philosophical question of the place of mathematical verification of the properties of a program, or whether a program can be verified by means of an algebraic proof. We assume that such attempts have value, even if only to aid the programmer in building up a coherent picture of the intended behaviour of the system.

with the predicate true. A proof consists of progressing backwards through the program, ensuring at each stage that the appropriate weakest precondition is satisfied before execution of each instruction. A loop is proved correct by showing that a given predicate (the "loop invariant") is true before, during, and after each loop iteration. The demonstration that an invariant relation holds is thus essential to this method.

Invariants are assertions about the program which hold in a sequence of states, but procedural programs require reasoning about the semantics of the actions involved in making transitions from state to state in order to assert that each state has some property, since there is no explicit relationship between one state and the next. Typically there will be more than one step necessary to reach a new state in which the invariant holds, and the invariant will not hold in intermediate states. The weakest precondition method allows for this need for several transitions to re-establish invariants by using loops as the operations over which invariants must hold, and then permitting more than one operation (e.g. multiple assignments) to be performed during each iteration of the loop. Reasoning about invariants over a number of states is a non-mechanical task, since invariants in a procedural program can only be shown to hold in a sequence of states by the use of some deep knowledge about the algorithm. For example consider the following wp program (cf. [Gries 81, p.199]):

```
i := 0 ;
sum := 0 ;
count := 0 ;
do
  even(b[i]) € (i≠n) §     sum := sum + b[1] ;
                           count := count + 1 ;
                           i := i + 1            (*)
  -
  odd(b[i]) € (i≠n)  §     i := i + 1
od ;
```

If the loop is to determine the average value of the even values in array `b[0:n-1]`, i.e. terminate in a state where the postcondition

```
{ R: (average = sum / count)
     € count = (Nj : 0 ≤ j < n : even(b[j]))
     € sum = (▪j : 0 ≤ j < n € even(b[j]) : b[j])          }
```

is true, then the loop invariant derived by replacing a constant with a variable might be:

```
{ P: (average = sum / count) € (i < n)
     € count = (Nj : 0 ≤ j < i : even(b[j]))
     € sum = (▪j : 0 ≤ j < i € even(b[j]) : b[j])          }
```

This loop invariant requires that the relationship between the value of `average` and the values of `sum` and `count` be maintained during execution of the loop, which requires the addition of an assignment to the end of the first command (at position `*`) to reassign the correct value every time that `sum` is changed.

A more natural method for maintaining a loop invariant relationship of this form would involve the use of a definition of the form

```
average = ( sum / count )
```

before the execution of the alternative command (the `do` loop), so that the value of average is maintained automatically.

In a definitive program it is a simpler task than in a procedural program to show that an invariant holds, since all functional definitions are maintained automatically. It is therefore easier to reason about the behaviour of a definitive than a procedural program using invariants, since the maintenance of the relationship is performed by the programmer in a procedural program.

§8.2.3 Parallelism

In any state-based model of computation progress is made by changing the state in some way. This is done by assignment in a procedural program. When an assignment takes place, the expression being assigned is evaluated. If several assignments occur in parallel then dependencies are created. The variables in the expressions must be evaluated in a way which avoids interference, i.e. by not allowing changes to a variable whilst it is being read. The principal advantage offered by the use of definitive programming, as described in §2.1.3, is that the parallel redefinition of a set of variables does not, in general, cause interference.

The adm model of computation allows a large amount of parallelism. The guards can all be evaluated in parallel, since no values are changed during guard evaluation. Once certain checks have been performed the commands in the run set can be executed in parallel. The checks which must be performed appear to be simpler than those which

would be necessary in a procedural environment. A procedural assignment of an expression requires each variable in the expression to be evaluated. It will in general be computationally expensive to test for interference between assignments, because any evaluated variable could cause interference, through being simultaneously assigned a value and evaluated. Parallel redefinitions, however, can be performed concurrently without interference provided the same variable is not being assigned to twice. We do not suggest that the problems of interference have been totally avoided, merely that we have rationalised the points in execution where they need to be detected.

§8.3 Comparison with procedural and nonprocedural programming

In this section we compare definitive programming with other approaches to programming. This is done by considering the components of a definitive program and contrasting these with procedural and nonprocedural programming concepts.

A definitive program combines procedural and nonprocedural programming elements. The procedural component is the state-transition model on which computation is based. The nonprocedural element comes from the use of definitions, which establish relationships between components of the system. In a definitive program there will be a combination of procedural and nonprocedural elements, based on the semantics of the underlying application. There is no prescribed method for creating this synthesis, which can be contrasted with the admonition to apply the principles of other paradigms, such as "strong data typing" in procedural programs and "modularity" in object-oriented programming.

The exact combination of procedural and nonprocedural elements will arise as a result of the applicability of definitions for the representation of the algorithm and data used. Applications will vary immensely in how much interdependence is exhibited by elements in the system. On the one hand the application might involve modelling the behaviour of a container filled with particles whose motion is random [Abelson & Sussman 85], in which case there are no apparent functional dependencies between any components of the system. On the other hand one might wish to model the interior workings of a complicated mechanical timepiece, in which case there are functional dependencies between nearly all components of the system. The ramifications of actions can be appropriately modelled at all levels using a definitive approach. If the action is local and insignificant, then no other variables need be updated. If on the other hand the action, though local, has implications at a higher level (e.g. a handshake

between honest people), then this can be effectively modelled by the use of definitions.

Nonprocedural programs describe in a clear way the functional dependencies within a system, whereas procedural programs are good at representing the current values of elements in the systems. In a functional program the functional dependencies are all characterised. It is not possible to perform a globally inconsequential local action without totally re-evaluating all functions. As an example, Miranda [Turner 85] includes a definition feature, whereby one function can be defined in terms of another. The environment established by a script cannot be modifed during function evaluation, which means that the only way to make a small change to the systems of definitions is to edit the script which gives the definitions, and then repeat the function evaluation. This is because of the lack of any form of history sensitivity [Backus 78, p.614] in functional languages.

Procedural programs are not as open to formal analysis as nonprocedural programs. In a procedural program the state at any point during execution is given by the values of all variables. Using definitions instead of values gives a more expressive model of state than simply assigning values to variables. Not only are values described, but also relationships between variables. Functional relationships in a procedural program are only implicitly described, and can be inferred only by means such as examination of the source code. Procedural programs do, however, allow for small changes of state without the need to perform a complete re-evaluation of the state. This is a particularly attractive feature for applications that entail incremental development, where progress is made by minor transitions from state to state.

Definitions characterise links between elements of the problem being addressed, permitting a more intuitive approach to programming. Maintaining definitions, which are often found in the specification of the problem, is complex in a procedural framework. In §1.4.1 we showed why going from a specification to a procedural program is problematical - it will in general be an easier programming task to go from a specification to a definitive program than to a procedural program. This is because more work (in the form of maintaining these underlying definitions) is being done by the language compiler, thus allowing the programmer to concentrate more on other issues of the problem. Similarly, it may be easier to mechanically show that assertions hold in a definitive than a procedural program, since all relationships are maintained in a definitive state.

The hope is to capture both the comprehensive representation of functional dependencies and the incremental nature of transitions within a definitive framework, by allowing the judicious application of the procedural notion of

transition within an environment which supports dependency maintenance. Functional principles can be applied where functional relationships actually exist: there is no obligation to impose relationships on an application which does not naturally exhibit them, as there often is with nonprocedural programming.

The state of execution of an adm program is given by the contents of the definition store and the action store. We therefore view the "program" part of a definitive program as data. From the state the next transition which will be made can be inferred. This differs from other forms of programming, where the state only defines the current values of variables, but does not indicate what will be the next transition. The definition store D contains that part of the state which can be evaluated, and the action store A contains the currently extant actions.

§8.4 Extensions to the adm

We have considered the applications of the adm, and compared definitive programming with other approaches. We now look to the future and describe potential extensions to the definitive machine of Chapter 2. We consider these extensions here because we assume that the reader is now more familiar with the adm than at the end of Chapter 2. We introduce the use of a windowing interface to serve as an interface between the program and the programmer, and then extend the notion of permissible input to allow a superuser to interact with an executing program. We describe how the interaction of a superuser can be used to allow an asynchronous mode of execution.

§8.4.1 Windows

We can form an interface between the programmer and the program by the use of windows. The windows which we introduce in this section will be useful for two purposes: for debugging the adm program and to be used by the superuser described in §8.4.2. We use windows to display information about the current state of the machine. There are eight such windows.

The program window shows the contents of P. The action window shows the current contents of the action store A, and highlights guards which are true. The definition window shows the current contents of the definition store D, and can be configured to show either the definition or value (or both) of a variable. It highlights variables

which are undefined. The status window shows the messages generated by procedural actions, indicating when the end of each execution cycle occurred, and any fatal error messages.

We introduce two windows in which to report errors: the notifiable window for reporting of notifiable errors, and the avoidance window for the reporting of avoidance errors. A third window, the run window, is used to show the current contents of the run set. The final window is the input window, and this is used for the entry of input. The input consists of redefinitions of variables and dynamic instantiations and deletions of entities.

§8.4.2 Superuser

We can distinguish between three categories of use of `am`. We call the person that writes the definitive program which is entered into the adm the programmer. The programmer can use the windowing interface described in §8.4.1 to aid in debugging the program. We call the person that can enter commands to change the state when the execution has halted the user. In this section we describe a third type of `am` user: the superuser.

The superuser is distinguished from the programmer by interacting with the program during execution, and from the user by the extent to which she can interact with the program. The superuser can enter commands for execution at any point in computation, not just when execution has halted. At any point in the computation the superuser can interactively enter a command via the input window. The command will then be executed with any internally selected commands during the next execution cycle. The superuser can use the windows introduced in §8.4.1 to monitor the current state of execution of the program.

This provides a very powerful method of interaction. We can employ this method of input to elicit information from the superuser. As an example the following entity inputs and validates the age of the user, and can be compared with Figure 3.4:

```
entity get_age() {
definition
   input_age, age_prompt = false
action
   ¬age_prompt print("Please enter age") -> age_prompt = TRUE,
   input_age < 0 print("Age cannot be negative, please re-enter") ->
            input_age = @,
   input_age > 130 print("Nobody is that old, please re-enter") ->
            input_age = @,
   (input_age >= 0) € (input_age <= 130) ->
```

```
            age = |input_age| ; delete get_age()
}
```

The superuser can also disable the default classification of singular states and invalid transitions into notifiable, avoidance and fatal errors which was described in §3.1. The superuser can be an actual person or a programmed description. If a person then as each singular state or invalid transition is detected the superuser can resolve the problem interactively. If the superuser is a programmed description then it consists of the actions which are to be performed to resolve singular states and invalid transitions, and the commands which are to be input during execution.

The superuser can therefore influence how the abstract definitive machine deals with singular states or invalid transitions. By putting a control structure on an adm program we can also allow the superuser to control the nature of execution of the adm in a far wider sense. As an example, suppose we want to execute an adm program by allowing the superuser to select guards for evaluation. All guards of the form

$guard_n$ -> $command_n$

are changed to

$enable_n$ € $guard_n$ -> $command_n$, (1)
$control_n$ -> $enable_n$ = FALSE (2)

and for each guard the following variables added to D:

$control_n$ = $enable_n$
$enable_n$ = FALSE

Figure 8.1

When the superuser sets $enable_n$ to true using the input window both (1) and (2) are enabled. (2) causes both guards to be disabled on the the next iteration, which means that the original guarded command in (1) will be enabled for precisely one iteration, during which it may or may not be true. This technique enables the superuser to interactively enable a set of guarded commands for exactly one iteration, by entering a command in the input

window to perform a list of redefinitions, each of which defines an appropriate `enable` variable as **`true`**.

Similar techniques would allow enabling for two iterations, toggling the enabling, and so on. We describe a means to provide this facility automatically in §8.8. The superuser concept and the use of control structures such as the one illustrated in Figure 8.1 allow the execution of a definitive program in an asynchronous manner, as described in §8.4.3.

§8.4.3 Asynchronous mode of execution

In §5.4.1 we described how an LSD specification can be transformed into a family of programs. It may be that it is desired to closely control the execution of the resultant program, to the extent of specifying precisely which guards are to be evaluated in each execution cycle, which commands are to be executed, and what to do in an inconsistent state or before an inconsistent transition. This would allow simulation of the asynchronous behaviour described by an LSD specification in any way which was desired.

Instead of the use of the `rand()` function which is employed in the transformation process of §5.4.1 for selection of when to enable guards and execute commands we apply the control structure described in Figure 8.1 to every guarded command in the definitive program transformed from the LSD specification. This allows the superuser to dictate when guards from the LSD specification are selected for evaluation, and when commands in the LSD specification are executed.

In §5.4.1 a method was given for splitting actions up into a set of guarded commands using the `selected` and `level` variables. The resultant set of guarded commands contained one which corresponded to the original guard and one for each of the commands in the original command list. By applying this method to an adm program and adding the control structure of Figure 8.1 we can convert an adm program into one in which selection of each guard and execution of each command must be explicitly enabled by the superuser. Variations on this are possible, for instance the use of a control structure which allows more than one command from a single command list to be executed in a single execution cycle.

The addition of this control structure allows asynchronous execution of an adm program, which can be executed either under the control of a human or a programmed superuser. If the superuser is programmed then the strategy

used for deciding which `enable` variables are to be set to true will change the manner in which the program is executed.

If the programmed superuser uses a random strategy then the converted adm program will be executed in an asynchronous manner without the need for a human superuser to decide which guarded commands to enable on each execution cycle. Asynchronous execution can allow the detection of singular states or invalid transitions which do not arise when executing the adm program, because of the synchronous nature of execution introducing unintended synchronisations.

If the strategy is based on choosing commands for execution which are likely to lead to an invalid transition in the current or a future execution cycle then the adm program will be executed in a manner which tries to show if interference is possible. The detection of whether an adm program executed in this way can lead to invalid transitions requires a complex analysis. Consider two commands in the original adm program: C_1 and C_2. Suppose that C_1 is a redefinition which enables a guard which has associated with it a command which cannot be performed in parallel with C_2. An invalid transition can result from executing C_1 before C_2, and not otherwise. Other more complex scenarios can easily be envisaged.

A third strategy would be to enable a maximal subset of guards which do not lead to an invalid transition. This mode of execution allows computation to continue for as long as possible, by using the superuser to avoid invalid transitions. When there is a command which cannot be executed (e.g. because it performs an invalid dynamic action) then its associated guard is not enabled. When there are two commands which cannot be executed in parallel then a heuristic is applied to decide on one which is to be ignored. A suitable heuristic might be as follows: when two commands interfere with each other, they will consist of either two redefinitions, one redefinition and one dynamic action, or two dynamic actions. In the first case do not enable the guard of one of the redefinitions arbitrarily. In the second case do not enable the guard of the redefinition. In the case of two dynamic actions which interfere, they will either be the same action (i.e. both deletions or both instantiations) or different (one instantiation or one deletion). If they are the same, then do not enable the guard of one of them arbitrarily. If they are different, then do not enable the guard of the invalid command.

§8.5 Use of definitions in LSD

In this section we describe how definitions are used in LSD. Derivates are the only kind of variable with which definitions can be associated. The two ways definitions are used are associated with the two uses of derivates: local and global. A local derivate is one whose definition uses only variables which appear in the variable declaration section of the agent description. A global derivate is one whose definition uses a variable which is not declared by that agent.

When a local derivate is used, it plays the part of a macro as described in §8.1. It cannot be redefined once the agent has been instantiated, and so the definition associated with it is static. Wherever it is used, either in guards, as a parameter, or on the right hand side of an assignment or a state variable, its associated definition could be substituted. The definition is used to represent a relationship that is perceived by the agent

When a global derivate is used the authentic value of undeclared variables might need to be used (as discussed in §5.4.1). This requires close synchronisation between the agent owning the derivate and the agent owning the undeclared variable, as identified in §5.3.1. In this case the definition is used to represent the way that the agent can perceive the effects of some relationship underlying the model, even though unable to perceive the individual components of the relationship. An example of this form of definition would be `dark = is_blind ⁄ no_light`, which describes how a person can perceive whether it is dark or not, even though not being able to tell whether they have gone blind or there is no light to see by.

§8.6 Unresolved issues in LSD

In this section we indicate unresolved issues, with some speculative solutions, in the description of LSD given in Chapter 4.

In Chapter 4 we described the use of derivates in a manner that suggested that they would always be maintained at the correct value. This use of derivates as a mechanism for modelling may be an abstraction. For instance, when using a spreadsheet the variables on the screen which change as a result of an action by the spreadsheet user do so sequentially, rather than concurrent with the change by the user. This means that there will be short times when the

relationships between the variables on the screen is not maintained. However, the spreadsheet remains a useful tool because the sequential updates occur quickly enough for them to appear to be concurrent with the change by the user. This corresponds to synchronisation by assumption (§5.3.2). It is therefore useful, although fictional, for the spreadsheet user to consider the changes caused by their action to be concurrent. The maintenance of derivates can be performed within the adm because of the synchronous nature of its operation - the abstraction that they represent becomes clearer in an asynchronous, genuinely distributed, multiprocessor system.

We indicated in Chapter 4 that state variables are both known to and conditionally under the control of the owning agent. It may be the case that a state variable is conditionally under the control of the agent, but its value is not known to the agent. A light switch which controlled a light in another room would be one example where the agent can change the setting of the switch, but does not know if the switch is in the "on" or "off" position. Another example is the "open" button in the door controller example of Chapter 4: the person pressing the button is not aware of the internal value of the button, but can only observe the consequences of pressing the button (i.e. the door opening and closing). In particular if pressing the "open" button elicits no response in terms of the door opening, the user cannot know if it is the door opening mechanism which is breaking down due to an electrical fault (i.e. the value of `hold` is being set to **`true`**, but the action to open the door is not being performed) or if the press of the button is not being detected due to some mechanical fault in the button (i.e. the value of `hold` remains **`false`**).

It increases the potential for information hiding if we introduce a fourth kind of variable: state-oracle. A state-oracle variable would be one whose value is both known to and conditionally under the control of the agent, whereas a state variable would be one whose value was conditionally under the control of the agent but whose value was not known to the agent. In this case only the value of state-oracle and oracle variables of an agent would perceived.

In some sense the distinction in use is syntactic, since whether the value is used in the protocol can be inferred statically: the value of a variable is used if the variable appears in a guard, a procedural action, a parameter list for a variable or dynamic action, or inside an evaluation operator in a redefinition command. The use of the state-oracle notation, however, highlights the distinction between variables which are perceived and variables which are conditionally changed.

The introduction of a state-oracle kind would require the resolution of other issues. For instance, if an agent

owns a state-oracle variable then should it immediately perceive the value of the variable as having changed when it performed the command which changed the value ?

In §7.1 we described how no distinction is made in an LSD specification between an action not being performed because it would violate a "law" of the system being modelled and an action not being performed because the agent chooses not to perform the action. In an LSD specification we represent the actions of physical laws (in the world being modelled) and actions chosen nondeterministically in the same way. In the family of adm programs which are the result of transforming the LSD specification this is reflected by not distinguishing between an entity performing an action because it chooses to (e.g. opening the elevator door in Chapter 5) and because it has to (e.g. pushing a block in the example of Chapter 4 causing the block to change position).

We could specify that all derivates are to be transformed into variables defined by formulae referring only to authentic variables. This would ensure that derivate values are always as closely synchronised with authentic values as possible. We have left the choice of whether to use the authentic or perceived variable as a parameter of the transformation process. If a variable in the definition of the derivate is not perceived, then the authentic value is used.

We have not developed a formal method for analysing an adm program describing a potential behaviour of an LSD specification. An example of analysis of the behaviour of an adm program is given in §4.4, where we showed that the adm blocks program cannot lead to an invalid transition occurring in the run set. The method of considering which guards are mutually exclusive, which because of the synchronous nature of the adm indicates which command lists cannot be executed in parallel, can be used for any definitive program. It will be necessary to develop methods for analysing the behaviours which can be described by the LSD specification and conform to the semantics of the system being modelled, in order to show that appropriate interpretation of the specification does not lead to interfering behaviour. This will probably not be possible to do by analysing all the adm programs which can be derived from the specification because of the (potentially infinite) number of different values for parameters which can be used in the transformation process - perhaps indicating a limit to the value of simulation.

§8.7 Appraisal of the modelling and simulation method

In this section we describe some of the potential benefits arising from our method for modelling and simulating concurrent systems. We illustrate these benefits with reference to the telephone specification and implementation. We consider how execution can be interpreted in terms of the specification, the cognitive nature of errors indicated by the simulation, how the simulation can be hand-driven, and how the simulation can be animated.

§8.7.1 Simulation of an LSD specification

An agent in an LSD specification can be transformed into an entity in a definitive program. When an instance of an entity is executing its behaviour represents one way that the corresponding agent in the specification could act. It is possible to relate the behaviour of the entity to the LSD agent in a simple way.

In an entity there will be two sorts of guarded commands: those which are derived from the LSD agent and those which are used for control. For each guarded command in the LSD agent there will be several in the entity: one which selects the command and one each for each command in the command list. When a command has been selected for execution in the entity the variable `selected` is set to the number of that guarded command in the agent, e.g. in the telephone example of Chapter 7 when the user has decided to answer a ringing telephone `selected[_U,1]` is set to 5. The variable level indicates how many of the commands in the associated command list have been executed, so that `level[_U,1]` is 2 when a ringing telephone has been picked up but not yet spoken into.

This method allows the state of the simulation at any point to be viewed in terms of the original LSD specification. It is useful to present the results of the simulation in this form since this is how the modeller will be considering the results of simulation. A method for doing this is presented in §8.8.

§8.7.2 Analysing the results of simulation

Using the techniques of §8.7.1 and the output from the simulation (the procedural actions in the telephone example) the behaviour described by the specification can be observed. There are two reasons why this may not be the intended behaviour. If the parameters of the transformation process have been incorrectly set, then this may result in unintended interaction of entities or an inappropriate propagation of updates to variables. Alternatively if the

specification is flawed then inappropriate behaviour may result from simulation of the specification. The first reason for inappropriate behaviour is corrected by changing the parameters of the transformation process, and the second by changing the specification.

When observing the behaviour of the simulation it is relatively easy to tell why it is not performing as expected, because of the ease of referring back to the high level LSD specification. For example, at one point in the development of the telephone specification the `connect()` agent initialised `engaged[S,D]` as `|ringing[D]` ¬`onhook[D]|`. This resulted in a simulation in which `connect(6489,7124)` was instantiated, 7124 was picked up to make a call, and then 6489 was connected to 7124, with the result that 7124 started ringing. Once this problem had been identified, the reasoning that led to the realisation that the line could become engaged after the connection had begun was concerned with the system being modelled, and not at all with what was happening in the simulation. Using our method of modelling and simulation encourages a cognitive approach to the construction of appropriate models of behaviour.

If the specification is correct, then the inappropriate behaviour is caused by the setting of parameters for the transformation process, and can be corrected by considering which variables are being updated at the wrong rate or which entities are operating at an incorrect rate. Techniques for aiding this task are described in §8.8. Again consideration of the behaviour is cognitive, rather than based on the values of variables, since typical questions asked will be "how quickly does the user need to know that the telephone is ringing ?" and "does it matter if it is not immediately detected that a connection cannot be made because the telephone is engaged ?".

Use of LSD for modelling and the abstract definitive machine for simulation of the behaviour of a concurrent system allows attention to be focussed on the cognitive, rather than programming, aspects of the simulation. Thus the requirements analysis can receive feedback more directly from the simulation than is normally possible, allowing the specification to be corrected as required.

Insight is gained into the modelling of a system using this method in two ways. The appropriate use of definitions requires in-depth consideration of what affects what in the system and encourages the discovery of the relationships underlying the behaviour of the system. The consideration of the parameters to be used in the transformation process focuses attention on how the entities in the system *do* act, rather than how they *can* act, which restricts the behaviours potentially described by the LSD specification to those that are realistic for the system

being modelled.

§8.7.3 Interaction with the simulation

The simulation we have described has executed autonomously, so that the behaviour described by the definitive program can be observed. It will sometimes be desired to have explicit control over the transitions that are performed. This can be accomplished by repeatedly loading the run set to see what commands can be performed on that iteration and performing a set of the permissible actions. Instead of performing actions it is possible to change the state of the machine by the redefinition of a variable or by performing a dynamic action, and then reloading the run set to see what commands are then enabled. Alternatively the control structure of Figure 8.1 can be applied to all the guarded commands in the definitive program, which allows the superuser to selectively enable actions on each execution cycle.

Such a use of the adm corresponds to a superuser deciding what commands are to be performed in the next transition. This can be performed when using `am`, and an example of this is given in Appendix 7.9, which demonstrates how the modeller can decide that the next action by user 10 at one point in the telephone simulation is to replace their receiver, allowing a check that the intended consequences of this action occur in the simulation.

This potential for intervention by the superuser on each execution cycle allows a very versatile form of simulation, since the superuser can perform actions at any point and observe their effects. This makes it easy to test out any potential interference explicitly, instead of hoping that the random nature of the simulation happens to cause the circumstances in which the modeller is interested.

§8.7.4 Animation of the simulation

The method of using procedural actions to provide output describing the state of the simulation is unsatisfactory in several respects. The results of the simulation are presented in text form, not the easiest form to immediately comprehend. It is also not an approach which is consistent with the method of programming the abstract definitive machine, since it relies on the sequential nature of execution of procedural actions in each execution cycle, and does

not attempt to explicitly link the output to the current state of execution of the program. What would be more consistent would be an explicit linking of the output of the program to the internal state of computation, as indicated at the end of §3.2. Finally the use of procedural actions is inherently history-based, because the meaning of a message like "`block a moved left`" in the blocks example of Chapter 4 can only be understood if the log of the execution is available, to determine for instance whether the string is taut. What would be more appropriate would be a state-based method of output which depicted the current state, instead of a history-based method which necessitates interpretation of context-dependent messages.

A more appropriate method of output has been developed for the blocks program in Appendix 5 by using DoNaLD (Definitive Notation for Line Drawing). This provides a means of displaying an animated picture of the blocks on the screen, where the picture is described by definitions. Changes in the definitions of the variables in the blocks program is reflected in changes in the picture displayed on the screen. Procedural actions are used in each execution cycle to output the current value of all variables which parametrise the picture, as described in Appendix 5.1. This output is read by the DoNaLD interpreter and used to update the picture on the screen, which gives a graphical depiction of the current state of the simulation, including whether the string is taut, the blocks touching, block a being pushed left, block b held, etc.

A similar method of presenting results could be used for the telephone example, so that for instance the lifting up of the receiver would be described by an animation of a telephone. The intent would be to link the state of the picture with the state of the simulation.

§8.8 A modelling and simulation tool

We conclude this chapter by showing how our observations about the transformation process could allow the development of a menu-driven modelling and simulation tool. There are three components of the approach described which can be packaged together into a unified tool: the means of entering the LSD specification, how it is transformed into a definitive program, and how the resulting program is executed. We consider each in turn.

Currently the LSD specification is stored as an ordinary text file. The entering of a specification would be aided by a specially written editor. This would allow a specified agent to be edited, list the agents which can perceive a

specified variable, show which commands perform dynamic actions on a specified agent name, and so on.

The process of changing parameters can be done more easily than at present if an appropriate interface is developed to allow parameters to be supplied for the automatic transformation of the LSD specification. An imagined example of such an interface is presented in Figure 8.2, which illustrates how the setting of parameters can be automated. It must be ensured that the options selected for parameters are reflected in the transformation process.

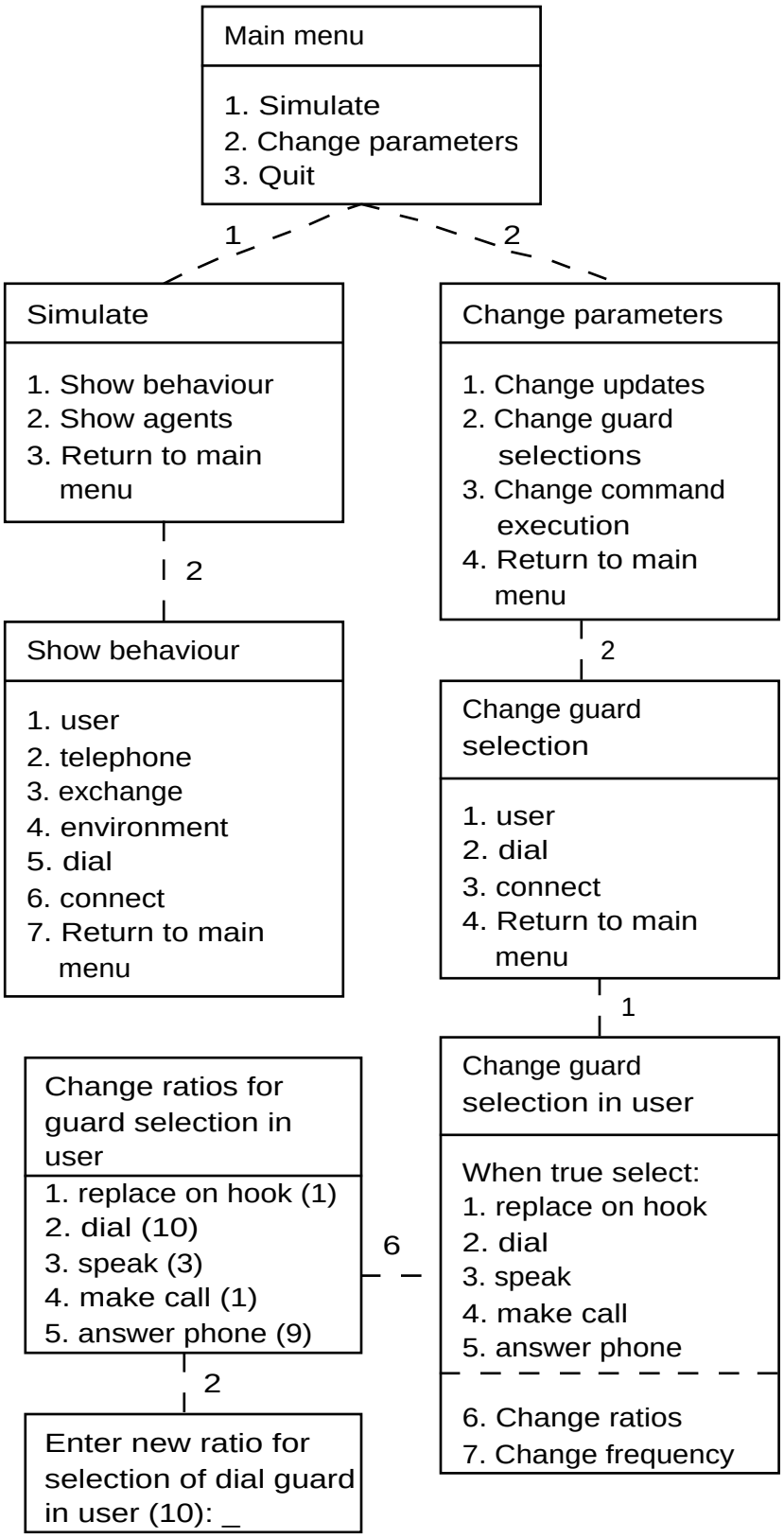

Figure 8.2

Finally we discuss how the results of simulation should be presented. The two variables `selected` and

`level` indicate where the simulation currently is for each agent instance, but at the moment this can only be inferred from their values, rather than explicitly represented. The explicit depiction of where the simulation is in terms of the specification would make it easy to see and follow exactly what the simulation is doing at any point.

We thus envisage an interface for the execution of the definitive program which can present the results of simulation at the same time as showing where the execution is in the specification. For example, when executing the telephone simulation it would be useful to see, with reference to the LSD specification, what each user is doing. When user 10 has just picked up telephone 6489 to make a telephone call this might be depicted using the symbol `*here*` as

```
agent user(10,6489) {
   ¬onhook[6489] § onhook[6489]= true ; dialled_number[6489]=@ ;
   ¬onhook[6489] € (tone[6489] == D) § dialled_number[6489]=N ;
   ¬onhook[6489] € (tone[6489] == C) § <speak> ;
   onhook[6489] € ¬ringing[6489] § onhook[6489] = false ;*here*
                                    dial(6489) ;
   onhook[6489] € ringing[6489] § onhook[6489] = false ; <speak> ;
}

agent user(11,7124) {
   ¬onhook[7124] § onhook[7124]= true ; dialled_number[7124]=@ ;
   ¬onhook[7124] € (tone[7124] == D) § dialled_number[7124]=N ;
   ¬onhook[7124] € (tone[7124] == C) § <speak> ;
   onhook[7124] € ¬ringing[7124] § onhook[7124] = false ;
                                    dial(7124) ;
   onhook[7124] € ringing[7124] § onhook[7124] = false ; <speak> ;
}
```

Figure 8.3

Notice how there is no indication of where user 11 is, which means that no guard has been selected. Later in the simulation there may come a point where user 10 has decided to speak and user 11 to replace the receiver. This would be shown by

```
agent user(10,6489) {
   ¬onhook[6489] § onhook[6489]= true ; dialled_number[6489]=@ ;
   ¬onhook[6489] € (tone[6489] == D) § dialled_number[6489]=N ;
   ¬onhook[6489] € (tone[6489] == C) § *here* <speak> ;
   onhook[6489] € ¬ringing[6489] § onhook[6489] = false ;
                                    dial(6489) ;
   onhook[6489] € ringing[6489] § onhook[6489] = false ; <speak> ;}
```

```
agent user(11,7124) {
   ¬onhook[7124] § *here* onhook[7124]= true ;
                          dialled_number[7124]=@ ;
   ¬onhook[7124] € (tone[7124] == D) § dialled_number[7124]=N ;
   ¬onhook[7124] € (tone[7124] == C) § <speak> ;
   onhook[7124] € ¬ringing[7124] § onhook[7124] = false ;
                                    dial(7124) ;
   onhook[7124] € ringing[7124] § onhook[7124] = false ; <speak> ;
}
```

Figure 8.4

The option to show this information can be selected from the interface described in Figure 8.2. Interactive use of the simulator is facilitated by the control structure of Figure 8.1, which can be used to allow the user of the simulation package to select which agent or agents is to executed next. This would be done by clicking on all the agents which are to perform their next action in parallel, and would provide a powerful but very easy to use mechanism for allowing the user to control the interaction of the agents in the system. If the agent is uncommitted, for example user 11 in Figure 8.3, then the guards which are true could be highlighted to allow one to be selected by the user, or a guard randomly selected by the simulator. This allows an implementation of the superuser to permit selection of which guards are to be evaluated. The run set can also be displayed in this manner to allow interactive selection of commands for execution.

This interaction between the user and the simulator would be facilitated by making the selection of an option by the user equivalent to the assignment of values to control variables. This puts the user and the simulator on a more equal basis, with the control of the simulation sometimes under the simulator, when performing the simulation autonomously, and sometimes under the user, when the next guards and commands to use are interactively selected by the user.

The potential for direct user intervention in the simulation in this way can be compared with the use of a debugger. It is, however, more powerful than a simple debugger which allows the execution to be traced line by line, in that the user actually takes an active role in the execution of the adm program, in selecting guards and commands.

The interface of Figure 8.2 could be made more user-friendly, with pull-down menus, single keystrokes for moving between menus, etc. If the "Simulate" option is chosen then the specification is transformed if necessary (i.e. if parameters have been changed since it was last transformed) and execution commences, with results being

shown according to the options selected in the "Simulate menu". The describing of an animated picture in terms of variables in the simulation might be another option in the "`simulate`" option of Figure 8.2, which would use DoNaLD to create a graphical representation of the behaviour of the program. Possible other options include editing the LSD specification and restricting the output of the simulation to only include certain messages. If the option to edit the specification is chosen then the specification editor is invoked. If the output restriction option is chosen, then only certain procedural actions are performed. As described in §8.4.1, the windows associated with the adm can also be configured to show the most appropriate information.

When executing the definitive program it would be possible to examine and reconfigure the various windows, load the run set, continue simulation, change the number of execution cycles to be performed, continue execution until a specified guard becomes true, and so on. The option of continuing until a specified guard becomes true is particularly useful. In the telephone program it might be used to continue until a line is engaged and then examine how the `connect()` entity reacts, or until a telephone is picked up to see what commands are then put on the run set.

A useful option would be some mechanism to store the state so that the simulation can be re-started or repeated at a later time. This should not be too difficult to implement, since it would just require the storing of the program, definition and action store. This facility would make `am` more useful for simulations which take a long time.

Ideally all the facilities described in this section would be integrated into a single modelling and simulation package, which might make a significant contribution in the simulation market.