Chapter 3

Implementation Issues

A rational man acting in the real world
may be defined as one who decides
where he will strike a balance between
what he desires and what can be done

W. Lippmann. The Public Philosophy. 1955.

§3 Implementation issues

In this section we describe the implementation-dependent issues addressed and resolved in implementing a simulator for the abstract definitive machine. The simulator is called `am`. The program `am` is a simulator in the sense that the commands in the run set are performed sequentially, rather than in parallel. The term is not to be confused with the method for simulation described in Chapter 5.

Implementing a simulator for the adm involved the design of a language in which to write a definitive program and the writing of an interpreter to execute the program in accordance with the model of computation described in chapter 2. We consider how singular states and invalid transitions are dealt with by `am`, how input and output is supported, how definitions should be stored, and what computation to perform during an execution cycle.

§3.4 is not essential for understanding the operation of `am`, and the description of the execution of an `am` program which is given in §3.5 illustrates how the material in §3.1 affects the description of execution given in §2.2.2. We give two short examples of adm programs in §3.7, and a large worked example in Chapter 4.

§3.1 Handling singular states and invalid transitions in `am`

In this section we consider how `am` deals with singular states and invalid transitions when they arise during execution. We call the occurrence of a singular state in D or an invalid transition contained in the run set an error. We do not require user intervention whenever an error occurs. Instead we introduce default actions which are taken by `am` when an error is detected.

We partition errors into three classes: notifiable, avoidance and fatal. Errors can occur for two reasons: variables not being evaluable or parallel actions interfering (i.e. singular states or invalid transitions). There are three reasons why a variable in D may not be evaluable: it is undefined, it is defined in terms of non-evaluable variables, or it is defined in a circular manner. The occurrence of a non-evaluable variable in D is depicted in Figure 3.1.

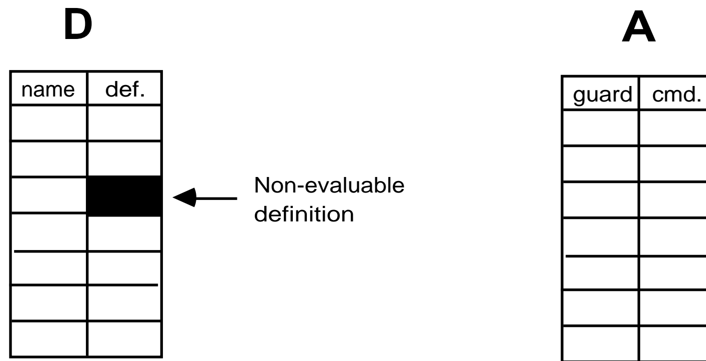

Figure 3.1 Notifiable Error

The presence of a non-evaluable variable does not necessarily indicate that the program is flawed. For example, when writing a prototype (§8.2.1) it is to be expected that not all variables will be defined. We report the fact that there is a non-evaluable variable in D as a notifiable error. The importance of notifiable errors depends on the application.

The guards in A are evaluated in a lazy manner (§2.1.1). If the value of a non-evaluable variable is required for evaluation of a guard then the guard cannot be evaluated. The variable may not be evaluable for the same reasons that a notifiable error is issued about the variable or because it does not appear in D. The occurrence of a non-evaluable guard is depicted in Figure 3.2.

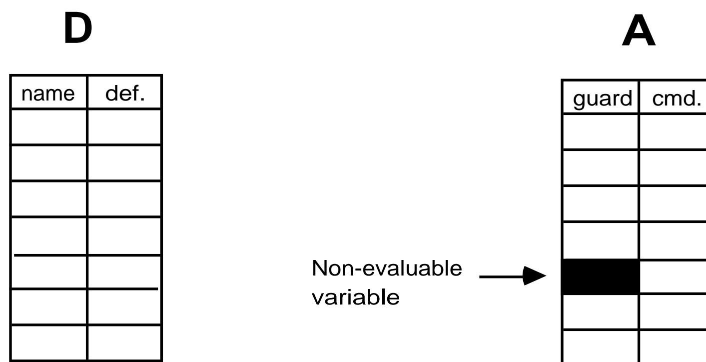

Figure 3.2 Avoidance Error

When a guard cannot be evaluated an avoidance error is issued. When an avoidance error is issued the guard

which caused it is treated as false. This allows computation to continue when some guards are not evaluable.

When a guard is true execution of the associated command may require evaluation of variables, i.e. when variables are evaluated in a redefinition (including procedural actions, described in §3.2) or used as parameters for dynamic actions or parametric variables. If any of the variables cannot be evaluated, as depicted by Figure 3.3, this is a fatal error.

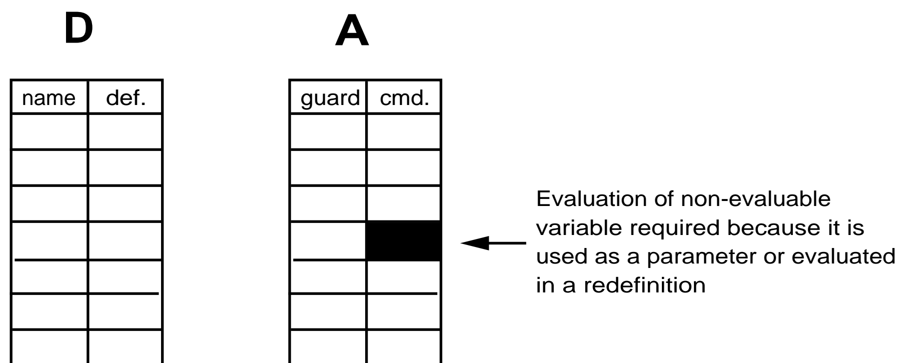

Figure 3.3 Fatal Error

Fatal errors can also occur because of commands in the run set. We refer to a variable which has a dynamic action performed on the entity owning it as undergoing a change of status. A fatal error occurs when a guard is selected which leads to any of the following on the run set:

- a redefinition of a non-existent variable
- an instantiation of an entity not in P
- two redefinitions of the same variable (1)
- a redefinition and a change of status for a variable (2)
- two dynamic actions on the same entity (3)
- an instantiation causing two variables of the same name to be in D
- an invalid dynamic action, viz instantiation of an already instantiated entity or deletion of an uninstantiated entity

The numbered points marked apply only to commands in different command lists in the run set. (1) is not an error

when the redefinitions occur in the same command list. (2) is an error if the redefinition and dynamic action are in the same command list and either the redefinition comes before the instantiation of its owning entity or after its deletion. (3) is an error if the dynamic actions are in the same command list and are invalid when performed sequentially. It is difficult to statically test for interference between commands involving parameters because of the difficulty of alias detection. When a fatal error is issued execution of the program stops.

Notice that we have not caused a fatal error to be issued when a variable which is evaluated in a command (or a procedural action - §3.2) is also changed by redefinition of it or any variable on which it depends (as described in §2.1.3). An example is the state

```
profit = costs - sales
sales = £100000
costs = £80000
```

in which a transition is made involving two redefinitions

```
year_end_profits = |profits|
sales = £100500
```

This transition would lead to interference if the evaluation and redefinition were performed in parallel, because the value of `profits` would be evaluated and changed at the same time. Another example (from Chapter 4) occurs when the run set contains the two commands

```
pb = | pb |
pa = | pa | + 1
```

where `pb` is defined as `pa + d`.

In `am` all evaluations within commands are done in the steady state in which the guard is evaluated, which effectively serialises two otherwise interfering commands into first the one involving the evaluation of the variable, and second the one involving the redefinition of the variable.

Before the run set is executed it is checked to test that it will not cause a fatal error. Only two of the above ways

that interference can occur do not involve dynamic actions. They are computationally very cheap to check, and so in a transition which does not involve any dynamic actions it is very easy to test for interference. When the transition includes dynamic actions it becomes more expensive to test the run set for potential interference, as is to be expected. The ability to check whether execution of the run set will cause a fatal error allows the user to intervene before the invalid transition occurs. When a fatal error is issued execution of the program stops and the user can take remedial action, by either redefining variables or performing instantiations or deletions of entities. It will in general be insufficient to simply re-evaluate the guards in the same state, because the deterministic nature of guard selection means that the same invalid run set will result. This means that no further progress can be made once a fatal error has been detected unless the user interacts to change the state.

The mechanism described in this section for handling singular states and invalid transitions is used in `am`. Other methods are discussed in §8.4.2.

§3.2 Output in `am`

The output device used in `am` is a screen, but the screen is used in the same way as a line printer, i.e. by printing the messages generated by `am` in a line-by-line manner. We therefore use the mechanism for output to a line printer described in §2.3.2. The `message` to be output is appended to the `output` variable by a command of the form

```
output = concat(output,|message|)
```

Since the command will always be of this form we can provide a better notation, which uses *procedural actions*. A procedural action is of the form

```
print (message)
```

and dispenses with the need to refer to the variable `output`. A message consists of a comma-separated list of elements to print, where each element is either a text string or an expression whose evaluation is to be printed. An example of a procedural action is

```
print ("Current index number: ",current_index)
```

which will cause the message

```
Current index number: 5642
```

to be displayed on the screen (if `current_index` evaluates to 5642).

A procedural action may involve the evaluation of variables (e.g. `current_index` in the previous example), so they must be performed in a steady state (§3.1). For this reason procedural actions are appended to guards, and performed whenever the guard is true. An example of a guarded command involving a procedural action is

```
new_book print ("Current index number:",current_index)
         § book(current_index) ;
           current_index = |current_index| + 1
```

A procedural action can be guaranteed to happen in each execution cycle by having a guarded command of the form

```
TRUE procedural action §          // empty command list
```

The only form of procedural action that has been investigated is the generation of messages. This is because `am` displays output on a screen. There is no reason to prohibit other types of procedural action, such as moving a robot arm or generating sounds, according to the output device being used.

An extension to this idea would be to view the entire screen as defined in the same manner as a definitive variable. This has been explored in the implementation of SCOUT [Yung 88], a definitive notation for SCreen layOUT. In the SCOUT system, the current value of the screen variable is what is currently seen on the screen. This corresponds to the other use of the `output` variable in §2.3.2 in which it is defined by a function in terms of the current state.

§3.3 Input in am

In `am` the user can perform commands when execution has halted, either because no guards are true or because a

stop command has been executed. When execution stops the user can instantiate an entity or redefine a variable. We can employ this method of input to directly request input from the user. As an example entity in Figure 3.4 inputs and validates the age of the user:

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

Figure 3.4

The entity is instantiated, and the correct input of the age is signalled by the variable age becoming evaluable. Notice the use of the evaluation operator, the way variables can be defined as undefined, and the deletion of the entity when it has set age to the appropriate value. When the prompt for input is printed and execution ceases, the user can define the variable input_age by entering a command of the form

```
define input_age = 34
```

The entity can be easily generalised to give a template which provides a mechanism for inputting and validating any sort of data. This use of templates for control structures can be heavily exploited in programs for the adm.

§3.4 How are definitions stored ?

In this section we consider how definitions should be stored. The adm has been implemented on a von Neumann architecture machine. Backus [1978, p.615] describes a well-known problem associated with the von Neumann architecture (§1.4.1). Backus calls the link which connects the store to the cpu the "von Neumann bottleneck", and notes that the purpose of a program is to change the store in some significant manner. The store can only be changed by the movement of data along the von Neumann bottleneck, and often what is being passed is not the required data

itself but reference information, such as the address in store of the required data.

The semantics of redefining a variable differs crucially from procedural assignment: there is no evaluation inherently associated with the redefinition. There are two methods we can use for storing definitions: either evaluate the value of the formula and store it at the same time as storing the definition itself or evaluate the variable whenever its value is needed.

The first method of storing the evaluation of the definition at the same time as storing the definition itself has the advantage that the value of the variable is immediately accessible, but also has two drawbacks. Each redefinition will then involve memory accesses to get the values of all dependent variables, computation time to evaluate the formula, and then two writes back to store: the redefinition and its evaluation. Writing a formula is likely to be more expensive than the procedural writing of a value, and the von Neumann bottleneck would therefore become further overloaded with the number of store transactions involved with this technique. The second problem is that variables may subsequently change value, which entails re-evaluation of dependent variables in order to keep their values consistent. This would involve some form of tagging mechanism to prevent a search of the entire set of variable definitions each time a variable is redefined. These tags would be used to record the dependent variables, i.e. those whose definitions refer to the variable. A redefinition would therefore involve a new definition and evaluation for the variable being redefined, an updating of all dependent variables, and an updating of dependence tagging information. Such an imposition is an unacceptable computational overhead whenever there are many redefinitions proportional to evaluations. This approach might therefore be suited to an application involving a small change to the state, and then large amounts of evaluation, as in the spreadsheet, where a redefinition necessitates updating the value of many of the variables on the screen.

An alternative approach is to evaluate the variable when required, and then maintain a consistent value for all variables which have been evaluated since these are the variables whose values are used in the program. This would still require the tagging overhead, and that dependent variables be re-evaluated in response to a redefinition. The savings in computation and passing information between the cpu and store is therefore minimal.

The second method for storing definitions is just to store the formulae and then to evaluate the variable on a need-to-know basis, i.e. whenever the value is required by the program. This method means that we no longer need to store dependency information and update the values of dependent variables after each redefinition, but requires the

evaluation of possibly nested definitions when the value of a variable is needed. This technique is therefore not suited to applications involving many more evaluations than redefinitions.

There is reason to hope that many programs would consist of several redefinitions of the same variable without intermediate evaluations being performed. This is often the case in procedural programs when assignment statements are used to maintain a consistent state, causing unnecessary evaluation since the actual value of the variable is not required in that state. We suggest that this redundant evaluation is caused by the nature of procedural assignments, and can be avoided by the use of definitions. The implementation of the adm evaluates variables as needed, although some applications which involve a proportionally large amount of evaluation will be more efficiently executed by using a strategy of evaluating at redefinition time.

To take full advantage of the nature of definitive programs we would need to use a special purpose (non von Neumann) architecture which is more suited to the execution of definitive programs. Such a suggestion is not without precedent: for instance the transputer architecture was developed as a result of insights gained by the use of CSP, and its programming language occam reflects this [May 87, p.2]. We briefly explore this theme for the rest of the section.

Traditionally the von Neumann architecture has been partitioned into two main components, the cpu and store. The store is seen as an passive component of the system which responds to requests from the cpu, by either reading or writing information. We suggest an architecture in which the store is an active participant in the computation process, with the task of maintaining consistency within the state.

If variables values are to be available immediately to the cpu, then there must be some mechanism to ensure that all dependent variables are updated after each redefinition. This is the responsibility of a processor in the store. When a variable is redefined, the redefinition is stored and the names of all dependent variables (kept as tags associated with each variable) are stacked. Whilst the cpu is involved in computation the variables on the stack have their values updated by the store processor. It is reasonable to expect that the cpu will not be accessing the store all the time, since it will have internal processing to perform as well, for instance the evaluation of a guard once the values of all the variables in it have been received from store.

When a read operation is invoked, the stack is checked to see if it contains the variable name. If it does, then the associated definition is evaluated and returned. If it does not, then the value associated with the variable is up-to-date, and so this is returned immediately, without the need for any evaluation. It is anticipated that the overhead of

stack checking will be compensated for by the gains in the time taken to perform evaluations. A read request thus asks for the value of the variable, not its definition, so it is now the responsibility of the store processor, and not the cpu, to perform evaluations. This machine may avoid the von Neumann bottleneck, since a write operation will only involve one memory access, with no intrinsic read operations as found with value assignments. It is also consistent with how the hardware of a digital processor actually works: the voltage at any point in a circuit is a function of the inputs to the circuit, and so processing at the lowest level of computation (i.e. circuit level) can be viewed as definitive. The area of hardware development for definitive programming might be a rewarding subject for future research.

§3.5 The `rand()` function

The `rand()` function is a function which takes one argument **n**, and returns a random number between 1 and **n** inclusive. It can be used in a definition, for example

```
x = rand(10)
```

The value of `x` which will be returned by evaluation will be a random number between 1 and 10. Different evaluations of `x` in the same definitive state may return different values. The redefinition

```
x = | rand(10) |
```

will result in the formula associated with `x` being a random number between 1 and 10, so that evaluation of `x` will always return the same number in the same defintive state. The `rand()` function can also be used in guards to provide a facility for randomly enabling a guard. An example of this is

```
(rand(2) == 1) § x = | x + 1 |
```

The introduction of the `rand()` function means that it is no longer the case (as stated in §3.1) that when execution of a program stops because no guard is true there is no point in re-evaluating the guards without changing the definitive state in some way; one of the guards in A might use the `rand()` function, and be true if re-evaluated.

§3.6 What does the execution cycle involve ?

In this section we describe an algorithm for the computation which ensures that all errors are detected and reported. The algorithm describes the execution involved in a single execution cycle, and is repeated until the execution is halted from within the computation.

```
begin
    while there is another variable to be examined in D                    test for notifiable errors
        if the variable is defined in a non-evaluable manner
        then issue a notifiable error
    while there is another guard to be examined in A                       evaluate guards,
        begin                                                               testing for avoidance errors
            evaluate the guard
            if the guard cannot be evaluated
            then issue an avoidance error
            else
                if the guard is true
                    then
                        begin
                            if      (the procedural action involves a non-evaluable variable)
                                or  (a non-evaluable variable is evaluated in a command)
                                or  (a variable used as a parameter cannot be evaluated)
                            then halt execution due to a fatal error
                            add the associated command list to the run set
                        end
        end
    if the run set is empty                                                 stop execution when no guards are true
        then halt
    if      (the run set contains a redefinition of a non-existent variable)    test whether run set
        or  (the run set contains an instantiation of an entity not in P)       contains invalid transition
        or  (the run set contains two redefinition of the same variable which are not
                    in the same command list)
        or  (the run set contains a redefinition of a variable and a dynamic action on the entity owning
                that variable, other than when the redefinition and dynamic action are in the same
                command list and either the redefinition comes before an entity deletion or after an
                entity instantiation)
        or  (the run set contains an invalid dynamic action)
        or  (the run set contains two dynamic actions on the same entity, other than when they are in
                the same command list and valid when performed sequentially)
```

then halt execution due to a fatal error
simulate the parallel execution of all the command lists in in the run set *execute commands*
end

User documentation for `am`, the implemented simulator for the abstract definitive machine, is contained in Appendix 3, and maintenance documentation in Appendix 4.

§3.7 Example programs for `am`

As a prelude to the description of a large program for `am` in Chapter 4, we give two small example programs in this section. The program in Figure 3.5 sounds the bell on a terminal after a prescribed period of time. As described in the user documentation in Appendix 3, parameters for `am` programs must start with a "_":

```
entity clock(_setting)
{
definition
   time = 0, time_set = _setting
action
   true print("Time is now ",time) -> time = |time| + 1
}

entity alarm()
{
action
   (time == time_set) print("^G") §                    // control-G sounds bell on terminal
}
```

Figure 3.5

The program of Figure 3.5 is executed by entering the two entity descriptions whilst in command mode (see user documentation in Appendix 3 for explanation of this term), instantiating the two entities with the commands

```
   clock(20)
   alarm()
```

(which indicates that the alarm should be sounded after 20 time units), and then starting the simulation with the command

```
   start
```

The program in Figure 3.6 computes the greatest common divisor of two integers.

```
entity one(_first)
{
definition
   var1 = _first, change1 = (var1 > var2)
action
   change1 -> var1 = |var1 - var2|,
   !change1 && !change2 print("gcd is ",var1) -> delete one(_first)
}

entity two(_second)
{
definition
   var2 = _second, change2 = (var2 > var1)
action
   change2 § var2 = |var2 - var1|,
   !change1 && !change2 -> delete two(_second)
}
```

Figure 3.6

The instantiations supply the numbers for which the greatest common divisor is required, e.g. to find the greatest common divisor of 24 and 16:

```
one(24)
two(16)
```

and then execution started by the command

```
start
```

This results in the following output:

```
gcd is 8
```