Chapter 2

Definitive programming and the abstract definitive machine

§2 Definitive programming and the abstract definitive machine

The aim of this chapter is to explain what is meant by the term definitive (definition-based) programming and to introduce the abstract definitive machine (adm). Definitive programming is described with reference to state-transition systems. The adm is introduced as an abstract model of computation for definitive programming in this chapter, and an implementation of the adm is described in Chapter 3. The abstract description of the adm in this chapter should be read in conjunction with the two examples of small adm programs in §3.7, the large example given in Chapter 4, and the user documentation in Appendix 3.

§2.1 What is definitive programming ?

In this section we describe what is meant by definitive programming. This is done by showing how a definitive notation can be used to describe the states and transitions in a state-transition system, and then describing how a definitive program is related to a definitive notation.

§2.1.1 Systems of definitions can model state-transition systems

Models of computation are often described in terms of state-transition systems. The state is described, and transitions change state to a new state. For example, a procedural program prescribes changes from state to state by assignment of new values to variables, which can be represented by a state machine [Minsky 72]. We describe our computational model with reference to state-transition systems.

In our model of computation we use *definitions* to describe state. A definition is of the form

```
variable = formula
```

and causes the `formula` to be associated with the `variable`. The `formula` establishes a functional relationship between variable values. This relationship is expressed using operators from a chosen underlying algebra, and applied to constants and the values of other variables. The `formula` represents a recipe for determining the value of the `variable`, and means that changes in the values of variables in the `formula` will in general affect the value of

the `variable`.

An example of a definition which describes when a car can be started is given by

```
can_start = hot ∧ choke
```

The relationship is between the values of variables, so if the value of `hot` or `choke` changes then the value of the variable `can_start` will always reflect any such changes. A system of definitions is used to describe a state. By way of illustration Figure 2.1 describes a state in which a car cannot be started because it is too cold and the choke is not in use.

```
can_start = hot ∧ choke
hot = temperature ≥ 60
temperature = 57
choke = false
```

Figure 2.1

The algebra selected for the variables in Figure 2.1 uses boolean and scalar data types, with the normal logical and arithmetic operators.

We call a state described by a system of definitions a *definitive state*. By this term we are not distinguishing a class of states as being definitive, but merely indicating that the state is represented by the use of definitions. We specifically preclude the possibility of there being two indistinguishable variables (i.e. with the same name) in a definitive state.

In a definitive state the evaluation operator can be applied to variables. This evaluates the value of a variable by using its associated formula as a recipe, and returns either a constant of the algebra (if the variable can be evaluated) or "undefined" (if it cannot be evaluated). Evaluation is by reduction: if variables used in the formula are themselves defined by formulae then they are substituted by their formulae, and so on until all variables in the formula being evaluated are defined by literal values. At this point the formula can be evaluated. In the above definitive state, application of the evaluation operator to `can_start` performs the following sequence:

```
can_start   = hot ∧ choke
            = (temperature ≥ 60) ∧ false
            = (57 ≥ 60)
```

```
= false
```

i.e. application of the evaluation operator to `can_start` returns the value **`false`**. The evaluation operator operates in a lazy manner, so evaluation of the formula "**`true`**`\x`" will always return **`true`**, regardless of the result of evaluating `x`.

We use the symbol `@` to indicate undefined. Evaluation returns undefined when the formula cannot be evaluated, which may be for a number of reasons. The formula is undefined if it is not supplied, refers to undefined variables, or is defined in a circular manner (i.e. directly or indirectly references the variable being defined). An example of a directly circular definition is "`a=a`". An example of an indirectly circular set of definitions is

```
happy = friends
friends = fun_to_be_with
fun_to_be_with = happy
```

This set of definitions reflects the way a person is happy if they have friends, will have friends if they are fun to be with, and are fun to be with if they are happy. The variable `happy` cannot be evaluated; application of the evaluation operator to `happy` would return `@`. Notice that the cyclic nature of a set of definitions may not be readily apparent, i.e. they need not be of the form "`variable = variable`".

We refer to a state in which not all variables can be evaluated as an *singular state* [Beynon et al 89a, p.8], which indicates that the state is not well-defined. We discuss singular states further in §2.3.4, §3.1 and §8.4.2. A variable `a` in a definitive state can be described either by the formula associated with `a` or by the result of application of the evaluation operator to `a`, i.e. by its value.

A definitive state is characterised by the definition associated with each variable. A transition to a new definitive state is caused by *redefining* a variable, which means changing the formula associated with it. In the system of definitions described in Figure 2.1, the redefinition

```
choke = true
```

causes a transition to a state in which `choke` has the formula "**`true`**" associated with it. In this new definitive state application of the evaluation operator to `can_start` returns **`true`**; `can_start` changes value even though its

definition has remained the same.

If the redefinition "`choke = choke`" was performed in the definitive state described in Figure 2.1 a directly circular definition of `choke` would result. If the redefinition "`choke = can_start`" was performed, an indirectly circular definition of `choke` would result. In either case, application of the evaluation operator to `choke` in the resulting definitive state would return `@`.

Evaluation can be used in the formula part of a redefinition. For example, the redefinition:

```
new_currency = | exchange_rate | * amount
```

results in the formula which is associated with `new_currency` being

```
new_currency = 5.22 * amount
```

if the current value of `exchange_rate` is 5.22.

Since we allow the possibility that some variables in the definitive state will return `@` when evaluated, we allow the explicit undefining of a variable by a redefinition of the form

```
variable = @
```

The formula associated with a variable can change without its value changing. For example, in the state of Figure 2.1 the redefinition

```
can_start = hot ~ choke ~ (temperature ≥ 70)
```

will not change the value of `can_start` in the resulting definitive state. Even though evaluation of any variable will still return the same value, the transition still results in a new definitive state.

When the value of `hot` in Figure 2.1 changes the result of evaluation of `can_start` will also change. This introduces the potential for changing the value of more than one variable in a genuinely concurrent manner, a feature which distinguishes definitive programming from procedural programming.

§2.1.2 Definitive notations

We require a language with which to describe the states and transitions of a system which uses definitive states and redefinitions to effect transitions. We call such a language a *definitive notation*. The syntax of the language is:

```
statement ::= evaluation | redefinition
evaluation ::= ? variable
redefinition ::= variable = formula
```

A statement is either an interrogation by evaluation or a redefinition. Evaluation returns the result of application in the current definitive state of the evaluation operator to the `variable` specified. Redefinition causes the association of a new `formula` with the specified `variable`. A redefinition causes a transition to a new definitive state.

As an example, consider the following definitive state:

```
a = b * c
b = 4
c = 2
```

The underlying algebra chosen for the definitive notation uses scalar values and operators. Two possible transitions from this state would consist of the redefinitions

```
a = b + (2 * c)        (i)
```

or

```
b = 6                  (ii)
```

Redefinition (i) will not change the value of a, so application of the evaluation operator to a will still return the

same element of the domain (i.e. 8) in the new definitive state. After redefinition (ii) the value that is returned by evaluation of a will be different, i.e. 12.

§2.1.3 Definitive programming

For the purpose of this thesis we adopt the following definition for a program for a state-based model of computation: a program is a description of transitions between states, with the transitions being performed automatically. A definitive program is one which allows redefinitions to be performed autonomously to cause transitions between definitive states. When a definitive program is executed sequences of states are passed through in an autonomous manner. A definitive program must specify when and what redefinitions are to be performed.

When a transition is performed will be context-dependent, i.e. it will be appropriate to perform some transitions in one state but not in another. For this reason a definitive notation does not provide a sufficiently expressive language for definitive programming. We require the ability to specify enabling conditions for transitions. This can be expressed using guarded commands. A guarded command consists of a guard and a command list. The guard is an enabling condition and the command list describes the transition to be performed if the condition is met. We call such a guarded command construct an *action*. An action is of the form

```
guard § command_list
```

In an action the enabling condition (i.e. *guard*) for a transition is evaluated in the current definitive state, and dictates whether the transition (i.e. *command_list*) is to be performed. Consider a definitive program which was in the definitive state described by Figure 2.1. An action which might occur in such a program is

```
¬can_start § choke = true
```

When the enabling condition for a transition is met (i.e. the guard is true in the current state) we may want to perform a number of redefinitions sequentially. This will mean that several states will be passed through as a result of executing one command. An example of an action of this form is

```
(time == 2000) § switch = false ; alarm = true
```

The command list in this example involves two redefinitions. The definitive state reached after the first redefinition (i.e. "`switch = false`") cannot be used for evaluating guards in - command lists are treated as a single transition, no matter how many redefinitions they involve.

Since the commands in a comand list are executed sequentially, an action only permits one redefinition to be performed at any time. If the guard of more than one action in a definitive program is true then this will result in the execution of more than one command list at the same time. This means that some redefinitions will be performed in parallel.

A central concept in this work is that the new state which results from the redefinition of a subset of the variables is in general independent of the order of redefinition. The only times when the order of redefinitions can be significant are when the same variable is redefined twice or when a formula in one of the redefinitions involves evaluation. An example of a redefinition involving evaluation is

```
rate_used = |exchange_rate|
```

which represents the fixing of the rate of exchange for a currency conversion. If the variable being evaluated (in this case `exchange_rate`) is also redefined, then the order of the two redefinitions is significant. This example corresponds to the use of an exchange rate at the same time as it is changing.

We call a transition which involves either the evaluation and redefinition of a variable or the redefinition of the same variable more than once an invalid transition, since it is not clear what state such a transition would result in. We discuss invalid transitions further in §2.3.4, §3.1 and §8.4.2.

A set of redefinitions which do not constitute an invalid transition can be performed in parallel. This contrasts strongly with the problems in parallelising procedural code which were identified in §1.4.

This observation motivates consideration of a computational model which can exploit this capacity for interference-free parallel redefinitions by using actions. The model was first presented in [Beynon et al 88b], and further discussed in [Beynon et al 89b].

The abstract definitive machine (adm) is an abstract machine which has been developed to give a computational model for definitive programs, by permitting more than one redefinition to be performed in a single transition. An adm program consists of a set of actions, and transitions are effected from state to state by executing the commands in command lists. Execution proceeds by evaluating all the guards, placing the commands with true guards in a run set, and then executing in parallel the command lists in the run set.

It must be ensured that the different commands in the run set do not constitute an invalid transition. To help guarantee this actions are organised into groups called entities, and a command is either the redefinition of a variable or the dynamic instantiation or deletion of an entity. This allows more control over potential transitions, since actions can be dynamically introduced or removed from the computation.

§2.2 The abstract definitive machine

In this section we describe the abstract definitive machine in more detail. We give a description of the structure and operational characteristics of the adm, and describe how it performs parallel redefinitions. In §2.3 we describe the implementation-independent details of using the abstract machine model to execute definitive programs, and in Chapter 3 we describe the implementation-dependent details of using `am`, the implemented abstract definitive machine.

§2.2.1 The program store, action store and definition store

The program store P is a static structure holding entity descriptions. An entity consists of a set of definitive variables in a definition statement and a set of actions in an action statement. Two examples of entities are:

```
entity alarm ()
{
definition
   switch = FALSE, alarm_time = 000000,
                                        //102431 means 10.24 and 31 seconds
   alarm = switch € (alarm_time == time)
action
   alarm § beep()
}
```

(where // indicates that the rest of the line is a comment) and

```
entity oven_thermostat ()
{
definition
   current_temperature, desired_temperature
action
   current_temperature < desired_temperature § power = TRUE,
                                                 //continue heating
   current_temperature > desired_temperature § power = FALSE
                                                 //stop heating
}
```

The alarm entity uses a definition to represent the way that the sounding of the alarm depends on the switch being on (i.e. true) and the alarm time matching the actual time. The oven thermostat entity uses guarded commands to turn on and off the power to an oven.

Entities in the program store can be dynamically instantiated and deleted during the computation. An `oven_thermostat()` entity is instantiated by a command of the form

```
oven_thermostat ()
```

and deleted by a command which uses the keyword **`delete`**, i.e.

```
delete oven_thermostat ()
```

We refer to the instantiation or deletion of an entity as a *dynamic action*. With reference to object-oriented programming, Meyer [1988, p.72] states "*objects are run-time elements that will be created during the system's execution; classes are a purely static description of a set of possible objects - the instances of the class*". We can draw an analogy between an instance of an entity in a definitive program and an object of a class in an object-oriented program.

The definition store D holds a set of definitive variables together with their definitions. Some variables will be undefined (i.e. defined as @), and have no formula associated with them. The variables in the definition part of an entity description are those which can only validly be referred to in computation when the entity is instantiated. Such variables are said to be *owned* by the entity. When an entity is instantiated the variables it owns are added to D. When the entity which owns the variables is deleted the variables owned by the instance are removed from D. The

contents of D therefore change dynamically during computation. An example of the contents of D might be

```
switch = TRUE
alarm_time = 073000
alarm = switch € (alarm_time == time)
time = 033245
```

As we have specifically precluded a definitive state from containing two variables of the same name (§2.1.1) we must ensure that instantiation of an entity does not duplicate variables in D. It is acceptable to delete one entity and instantiate another entity in the same transition which own a variable with the same name. This allows variables owned by different entities to have the same name and be referred to by omnipresent entities, provided that there is never more than one instance of an entity owning the variable.

The action store A contains a set of actions. Each action consists of a guard and a command list. A command list consists of a list of commands, each of which either redefines a variable or is a dynamic action, i.e. instantiates or deletes an entity. When an entity is instantiated the actions contained in its description in P are added to A. When an entity is deleted the actions associated with it are removed from A. The contents of A therefore change dynamically during computation. An example of the contents of A might be

```
current_temperature < desired_temperature § power = TRUE
current_temperature > desired_temperature § power = FALSE
switch_setting == 0 § desired_temperature = 0
switch_setting == 1 § desired_temperature = 150
switch_setting == 2 § desired_temperature = 200
```

or

```
alarm § ring()
¬switch § delete ring()
```

§2.2.2 Operation of the adm

Before executing an adm program the machine must be initially configured. This is done by instantiating a non-empty set of entities, so that sets of definitions are stored in D and sets of actions in A. The machine is then in its first state. The state of the machine depends on the contents of D and A. Evaluation can be performed in this state,

since nothing is being written to any store. Such a definitive state we will call a steady state.

All the guards in A are evaluated in this steady state. Any guards which are true have their associated command list stored in a run set. If the run set contains no commands (i.e there were no command lists with true guards) then execution terminates at this point. Command lists in the run set are then executed. If there is more than one command list in the run set then all the command lists are executed in parallel. Individual command lists are executed sequentially. This (for example) allows an entity to be instantiated and variables owned by it to be initialised in one command list, or permits an entity instance to perform some final commands and then delete itself.

If the command is a redefinition, then the appropriate variable in D is redefined. If the command is an entity instantiation, then the appropriate set of definitive variables and actions are copied from P into D and A respectively. If the command is an entity deletion then the variables and actions which were introduced when the entity was instantiated are removed from the D and A stores. If the command is the keyword "`stop`" then the remainder of the commands in the run set are executed and then execution terminates.

When all the commands in the run set have been executed the system is in a new steady state. The system has now completed one *execution cycle.* The process is then repeated.

§2.3 Programming the adm

We now describe how to program the abstract definitive machine. In this section we describe the implementation-independent details of the adm, which covers the parametrising of instances, output, input, and singular states and invalid transitions. These are characteristics of an abstract (i.e. implementation-independent) model of computation. A specific implementation of the abstract definitive machine is described in Chapter 3.

§2.3.1 Parameters

If entities are to be dynamically instantiated and deleted then it is important that instances of entities are distinct. We allow entity instances to be parameterised, so dynamic actions on entities supply both the entity name and a list of parameters. The supplied parameters are used to parametrise the variables associated with the instantiation. This

provides a mechanism for avoiding naming conflicts (i.e. two variables with the same name in D) when an entity is instantiated more than once. We specifically preclude the possibility of having two variables of the same name in the same definitive state (§2.1.1), and use parameters as disambiguating identifiers for the entity instances. It must be ensured that each instance of the entity has different parameters, since otherwise there would be indistinguishable entities simultaneously extant. This condition means that in the case of an entity which does not take parameters there can be no more than one instantiation of it at any time.

Parameters can also be used to pass information about the state within which the entity is instantiated, although this can also be done by taking advantage of the sequential nature of execution of a command list by defining variables owned by the entity after instantiating it. Parameters used in this way are passed in a call-by-value manner, i.e. the parameter cannot be redefined by the instantiated entity.

We allow variables to be parametrised by parameters used in the entity description. The program store description of an entity describing a book in a library might take the following form:

```
entity book (index_no) {
definition name[index_no], author[index_no]
}
```

The description of the entity in P will include a *formal* parameter list after the name of the entity, which lists the parameters that are required when a dynamic action is performed on an instance of the entity. In the `book()` entity the formal parameter list contains only one parameter, viz `index_no`. Variables in the entity description can only use parameters which occur in the formal parameter list.

When the entity is instantiated an *actual* parameter list of values is supplied, which are substituted for the formal parameters throughout the entity description in P before storing in D and A. For example, an instantiation of the `book()` entity by the command

```
book (264221)
```

would result in the variables `name[264221]` and `author[264221]` being added to the definition store. A deletion of this entity (for example when the book is removed from service) must use the same actual parameter:

```
delete book (264221)
```

We have only used literal values as parameters, but we can also use variables which are to be evaluated in the state in which the dynamic action is performed. For instance, the variable `current_index` might be used to record the next index number. The command to instantiate a `book()` entity would then be:

```
book (current_index) ; current_index ++
```

The value of `current_index` would be evaluated in the state in which the guard for the command was evaluated to true, and used as the parameter in the instantiation of the `book()` entity. The second command increments the value of `current_index`, so that its next use to instantiate a `book()` entity does not produce an instance with the same parameter. The command

```
current_index ++
```

is a shorthand for the command

```
current_index = | current_index | + 1
```

The evaluation operator is applied to `current_index` in the steady state in which the command's guard is true, and the result is used in the definition, so that the formula for `current_index` in D after execution of this command list is

```
264221 + 1
```

It is left as a programming task to ensure that parameters used are indeed distinct for each instantiation. The use of parameters increases the complexity of analysis of a program, because it introduces the procedural problem of alias detection. In particular, if two commands in the action store A are of the form

$guard_1$ § book(i)
$guard_2$ § book(j)

then it must be ensured that `i` and `j` do not have the same value if both commands are executed.

§2.3.2 Output

We have not discussed input and output so far. In this section we explain how output facilities are provided in the adm, and in the next section we describe input facilities.

A variable called `output` is used to provide output facilities. The variable is linked to the output device in such a way that changes to the variable cause changes in the state of the output device. The changes that are made to `output` should be consistent with the nature of the output device.

If the output device consists of eight LEDs, then an appropriate change to the value of `output` would be to define it by a formula which always evaluated to an integer between 0 and 255. The current value of `output` would at all times be interpreted as an eight digit binary number which indicated which LEDs are to be illuminated. This use of `output` involves defining it as a function of the current state, so that as the state changes so will its value and the state of the output device. The definition of `output` will be of the form

```
output = function of state
```

The use of LEDs does not entail the keeping of a history. Using a line printer as an output device means that the output device is not purely state-based, because it also involves the keeping of a log of what has been output. In the case of a line printer the redefinition of `output` must be procedural in nature, and consists of appending the new text to be displayed to the end of the current value of the `output` variable. The redefinition of `output` for this type of output device will be of the form

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

This will cause the value of `new` to be appended to `output`, and to be printed by the line printer. This use of `output` provides a record of what has been printed by the line printer.

§2.3.3 Input

Execution of an adm program terminates when either a `stop` command is executed or no further transitions can be made, i.e. all the guards in A are false. At this point computation ceases and the user of the program can make changes to the state. The user can perform the same actions as those found in command lists, i.e. redefinition of variables or the instantiation or deletion of entities. If execution of the program stopped because no guard evaluated to true then the actions performed by the user must cause at least one guard to become true. Execution of the program can then be resumed.

§2.3.4 Singular states and invalid transitions

In this section we describe how singular states and invalid transitions can arise when executing an adm program. A singular state is one in which a variable in D cannot be evaluated. In a system there may be some states in which a variable cannot be evaluated. It may be desired to model the system using the adm in a way that allows the representation of such states, e.g. when changing channel on a television which goes momentarily black whilst changing, in the state after the change has started and before it has been completed the television is not tuned to any channel. A singular state may also arise because of a flaw in the definitive program being executed.

An invalid transition is one consisting of commands which cannot be executed in parallel. Again a modelling of a system may involve the possibility of transitions which cannot realistically be performed in parallel, e.g. two people simultaneously pushing a revolving door in opposing directions. In such a case the resulting state is not well-defined. It would be inappropriate to regard an invalid transition on the run set or a singular state as automatically indicating a flaw in the program being executed, since it may indeed be a faithful modelling of the real world.

During the execution of an adm program different states will be passed through. In each state it is possible to tell if the state is singular and if the next transition will be invalid. Both a singular state and a state in which the next transition will be invalid can arise for different reasons (e.g. variable defined in a circular manner, duplicate redefinitions of the same variable, etc.), and the action (if any) which should be taken in each case may vary between applications. Using the concept of a superuser which is introduced in §8.4.2 it is possible to describe default actions to be performed when a singular state or invalid transition is detected. For instance, if the error is due to a singular state then we can choose to allow execution to continue if the non-evaluable variable will not affect execution, or we can allow the user to intervene to make the variable evaluable. Similarly an invalid transition can be dealt with by specifying beforehand ways of resolving the interference or by allowing user intervention to indicate either what

transition is to be performed or by changing the state to one where the resulting transitions are not invalid. It would be confusing at this stage to develop this theme in detail, so we describe the specific set of default actions implemented in `am` (the implementation of the abstract definitive machine) in §3.1, and consider a more general method of dealing with singular states and invalid transitions in the abstract machine model in §8.4.2.