Chapter 2

The Abstract Definitive Machine, ADM

The ADM is the Abstract Definitive Machine. The name itself makes a substantial claim. If we are to investigate the fundamentals of dependency and indivisibility, the ADM would seem a good place to start.

It is 16 years since the first writing about the Abstract Definitive Machine (ADM) was published. This chapter reviews the ADM in the light of subsequent research (see §2.5 for more details of the background sources consulted). A significant contribution of this review work has been to reveal subtleties and inconsistencies in the way in which the ADM concept has been presented and developed since it was first fully described by Slade in [Sla90]. Particularly significant is the fact that, since the early publications over the period 1988–1990 by Slade, Beynon *et al* that focussed on the ADM as an abstract machine model for parallel "definitive programming", the concept of the ADM as a *machine* has essentially been neglected. As evidence of this, Rungrattanaubol devotes a chapter of her PhD thesis [Run02, §3] to the "Abstract Definitive *Modelling* framework". During my critical review, it became clear that there is a significant difference between the framework described by Rungrattanaubol and the original ADM concept as described by Slade. The difference is significant enough to require (in my opinion) a totally new name for the framework as described by Rungrattanaubol (see §2.3.4).

In this chapter, we first attempt to clarify the concepts of the LSD account

and the ADM (in its original concept as described by Slade) and the distinctions between them. We then briefly review the existing implementations and describe implementation strategies that determine the organisation of Chapters 2, 3 and 4 of this thesis. In §2.3 we investigate the operational semantics of LSD and the ADM. A few small clarifications to the existing literature are found to be necessary. Section §2.3.4 then briefly reconsiders Slade's ADM as clarified and our requirements for an EM tool today, and identifies a discrepancy between the ADM as characterised in [Sla90] and Beynon and Slade's original intentions for use of the ADM in conjunction with LSD. This leads me to formulate an alternative algorithm for ADM execution for which the term 'Authentic' ADM is adopted. The context is then clear for a short investigation of the relationship between the ADM and Chandy and Misra's UNITY in §2.4. Finally an overview of the relevant sources is given in §2.5.

2.1 Concepts of the LSD notation and the ADM

2.1.1 LSD accounts

The ADM was primarily developed as a tool for the animation of LSD accounts. Most sources describe the ADM in the context of LSD. We must therefore briefly review the concept of the LSD account.

The LSD[1] notation (a Language for Specification and Description) [Bey87a] was originally conceived as a semantic model for the CCITT standard "Specification Description Language" (SDL). It was developed in 1986 by Meurig Beynon in collaboration with Mark Norris, then at the British Telecom Research Laboratories. LSD was intended to provide a medium for describing systems at high levels of abstraction, integrating functional and procedural models and synchronisation mechanisms. The first case studies were in fact of telephone systems, but the notation has since been applied in more situations and the underlying philosophy developed.

An LSD account does not have a formal operational semantics (hence the usual accompanying use of the word "account" rather than "specification"). It serves to describe aspects of interactions amongst agents that are identified in the preliminary understanding of a phenomenon or a domain. This corresponds to the usual starting

[1] In part so-called since LSD meant pounds, shillings and pence for its originator.

point of a system builder with an imprecise requirement, where the objective status of much of the domain is not yet known. The LSD account is intended to offer support at this initial, pre-operational, stage of understanding.

An agent description in LSD takes the syntactic form shown in Figure 2.1. The figure is the first attempt to be made in print of an in-depth description of the essential properties of LSD syntax. This is a difficult task as the syntactic conventions changed several times over the period 1986–1992 and previous sources have not made the distinction between the essential ingredients and any extensions[2] necessary in a particular domain and modelling context. I have constructed the figure from the more recent of the sources mentioned at the end of this section, and have taken the liberty of following Y.P. Yung's suggestion [Yun93, p.144] of renaming the term 'protocol' to 'privilege'. Yung does not himself follow his proposal, in order "to prevent too many versions of LSD notations being in circulation", but we feel now (10 years later) that clarity rather than conformity is the aim.

Using LSD, an agent may be described from the point of view of the modeller in the following terms (developed from [BCSW99]):

1. An identity, established by *agent_name* and the values of *parameter_list*;
2. `state` observables: Observables owned by the agent (in the sense that when the agent is absent, its state observables do not exist);
3. `oracle` observables: Observables that are deemed to influence the behaviour of the agent;
4. `handle` observables: Observables that the agent can conditionally (re)define during the course of an action;
5. `derivate` observables: Declared dependencies between observables[3] that are projected to hold in the view of the agent;
6. `privilege`s for action: Guarded actions that describe circumstances under which state-changing actions *can* (but not necessarily *will*) be performed.

[2] Examples include the use of types in Bridge's LSD specification of a vehicle cruise controller in [BBY92] and agent roles in Beynon *et al*'s LSD account of a digital watch in [BCSW99].

[3] These observables need not necessarily be `oracle`s of the agent.

agent ::= `agent` *agent_name* `(` *parameter_list_description* `) {`
[`state` *obs_name_list*]
[`oracle` *obs_name_list*]
[`handle` *obs_name_list*]
[`derivate` *derivate_list*]
[`privilege` *action_list*]
`}`

obs_name_list ::= *obs_name* (`,` *obs_name*)*
derivate_list ::= *derivate* (`,` *derivate*)*
action_list ::= *action* (`,` *action*)*
obs_name ::= *string*
agent_name ::= *string*

action ::= *guard* → *command_list*
guard ::= *boolean_formula*
command_list ::= *command* (`;` *command*)*
command ::= *redefinition* `|` *instantiation* `|` *deletion*
instantiation ::= *agent_name* `(` [*parameter_list*] `)`
deletion ::= `delete` *agent_name* `(` [*parameter_list*] `)`

derivate ::= *obs_name* `=` *formula*
redefinition ::= *derivate* `|`
obs_name `=` *formula_with_evaluation*

Notes:

1. The syntax of *formula*, *boolean_formula* and *formula_with_evaluation* are undefined, in order to allow for many possible type algebras. An exception to this is that the vertical bar '`|  |`' syntax is used to denote the evaluation of a sub-expression in *formula_with_evaluation* only.

2. The syntax of *parameter_list* and *parameter_list_description* are defined, but are omitted from this particular description.

3. An observable named `LIVE`*agent_name* represents the live-ness of the named agent.

4. The *obs_name*s exist in one "global name space" (to borrow a programming term). However, an observable is not observable/changeable by an agent unless marked as an `oracle`/`handle`.

5. In many contexts it is reasonable to assume for example that `state` implies `oracle` and `handle`, and that `derivate` implies `oracle` but no such implications are valid in general and so the status of observables should be explicitly stated in an account.

Figure 2.1: BNF describing syntax for an LSD account

Beynon *et al* [BCSW99] go on to describe the intended breadth of description, expanding on the note above about privileges representing *can* but not *will*:

> The LSD account is not to be mistaken for a formal specification of system behaviour. It is concerned only with how state-changing actions are attributed to agents, and how their interaction is mediated through observables at their interfaces. The context for agent interaction, and the viewpoint of an objective external observer are conspicuously absent.

The development of an LSD account of a phenomenon is intended to proceed in parallel with the construction of an EM model. Beynon [Bey97] describes the way in which a modeller's view of an agent may develop in the course of modelling, with reference to "three concepts of an agent within a unifying framework for model construction":

View 1: an entity comprising a group of observables with unexplored potential to affect system behaviour;

View 2: a View 1 agent capable of particular patterns of stimulus-response within the system;

View 3: a View 2 agent whose pattern of stimulus-response interaction can be entirely circumscribed and predicted.

Composing an LSD account allows us to more precisely describe our current understanding in terms of observables, dependencies and agency. Firstly we make a provisional decision about the number and identities of agents present — as represented by clusters of state observables. We then attempt to add detail to the account by classifying observables as oracles, handles and derivates and associating privileges to agents. The process of composing the account may itself suggest revisions, or experiments to be performed with the referent or EM model, which then provide further experience requiring revision of the account. Our understanding of agents may thus develop beyond view 1 perhaps as far as view 3, if the domain lends itself to such characterisation and our understanding is developed enough.

An excerpt from an example LSD account is shown in Figure 2.2, which is taken from [Yun93, Appendix H.1]. The figure shown is an excerpt from Y.P. Yung's account of a railway, including train, driver, guard, station-master, passengers and carriage doors. We will return briefly to this LSD example in §2.3.1.

This brief introduction to LSD will suffice here: for further detail, the development of LSD can be traced through [Bey87b] (=[Bey87a]), [BN87], [BNS88],

```
agent passenger((int) p, (int) d, (int) _from, (int) _to) {
// passenger p is intending to travel from station _from to station _to
// and he will access through door d of the train
state       (int) from[p] = _from;
            (int) to[p] = _to;
            (int) pat[p] = _from;
            (int) door[p] = d;
            (int) pos[p] = 2;
            (bool) alighting[p], boarding[p], join_queue[p,d];
oracle      (int) at, pat[p];
            (bool) queueing[d], pos[p], door_open[d];
handle      (int) pos[p], pat[p];
            (bool) door_open[d];
derivate    alighting[p] = at == pat[p] ^ at == to[p] ^ -2 ≤ pos[p] ≤ 0 ^ engaged;
            boarding[p] = at == pat[p] ^ at == from[p] ^ 0 ≤ pos[p] ≤ 2 ^ engaged;
            join_queue[p,d] = (alighting[p] ^ door_open[d] ^ pos[p] == -1) ||
                              (boarding[p] ^ door_open[d] ^ pos[p] == 1);
            LIVE = ¬(pat[p] == to[p] ^ pos[p] == 2);
privilege   boarding[p] ^ pos[p] == 2 -> pos[p] = 1;
            alighting[p] ^ pos[p] == -2 -> pos[p] = -1;
            alighting[p] ^ ¬door_open[d] -> door_open[d] = true;
            alighting[p] ^ pos[p] == 0 ^ door_open[d] ^ queuing[d]
                              -> pos[p] = 1; pat[p] = lat[; pos[p] = 2;
            alighting[p] ^ pos[p] == 0 ^ door_open[d] ^ ¬queuing[d]
                              -> pos[p] = 1; pat[p] = lat[; door_open[d] = false; pos[p] = 2;
            boarding[p] ^ ¬door_open[d] -> door_open[d] = true;
            boarding[p] ^ pos[p] == 0 ^ door_open[d] ^ queuing[d]
                              -> pos[p] = -1; pat[p] = at; pos[p] = -2;
            boarding[p] ^ pos[p] == 0 ^ door_open[d] ^ ¬queuing[d]
                              -> pos[p] = -1; pat[p] = at; door_open[d] = false; pos[p] = -2;
}

agent door((int) d) {
state       (bool) queuing[d], occupied[d], around[d];
            (bool) door_open[d] = false;
oracle      (int) pos[p], door[p]; (p = 1 .. number_of_passengers)
            (bool) join_queue[p,d]; (p = 1 .. number_of_passengers)
handle      (int) pos[p]; (p = 1 .. number_of_passengers)
derivate    queuing[d] = there exists p such that join_queue[p,d] == true;
            occupied[d] = there exists p such that (pos[p] == 0 ^ door[p] == d)
            around[d] = there exists p such that (door[p] == d ^ -1 ≤ pos[p] ≤ 1)
privilege   queuing[d] ^ ¬occupied[d] ^ join_queue[p,d] -> pos[p] = 0; (p = 1 .. number_of_passengers)
}
```

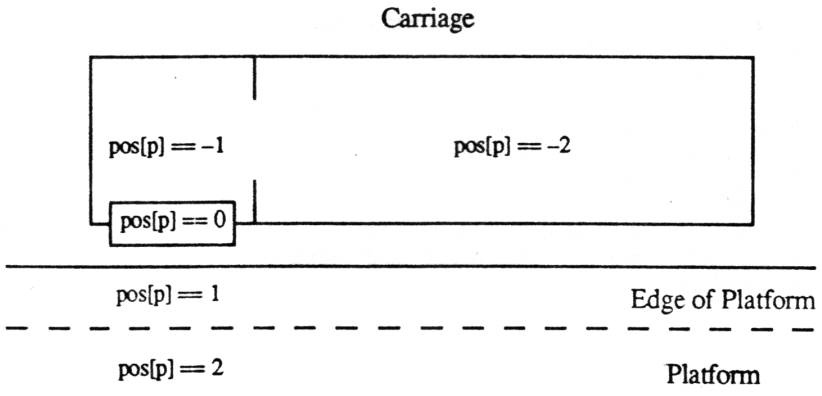

Figure 2.2: LSD account of a train carriage door and passenger (from [Yun93, Appendix H.1])

[BNOS90], [Sla90, §5.1], [BBY92] (which briefly describes the 'modern' version) and [Yun93, p.142].

2.1.2 1-agent modelling with the ADM

In [Sla90], Mike Slade presents the Abstract Definitive Machine (ADM), "a computational model for definitive programs". The ADM enables interaction with meaningful state through the provision of a *definitive notation*, which allows the formulation of a definitive script[4] to describe state and the introduction of redefinitions to make transitions manually.

The ADM uses *definitions* to describe state. A definition can be described with the following simple BNF:

definition ::= *var_name* = *formula*

It describes the association of the *formula* with the variable named by *var_name*. The formula establishes a functional relationship between variable values. This relationship is expressed using operators from a chosen underlying algebra. 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. Following Slade [Sla90, §2.1.1], a state described by a system of definitions is called a *definitive state*.

A *transition* to a new definitive state is caused by redefining a variable, which changes the formula associated with it. Redefinitions can take the same form as definitions, with an optional extension including evaluation.

redefinition ::= *definition* |
var_name = *formula_with_evaluation*

The same notation can therefore be used to describe the state and transitions upon the state. The current state is described by a set of definitions. A transition to another state is described by a set of redefinitions.

[4] Slade describes input to the ADM as a "definitive program" — we now prefer the term "definitive script" (a term not yet coined at the time of Slade's writing), which is intended to emphasise *state* over behaviour.

The current state as described by the set of definitions is 'timeless'. Although it is arrived at through a history of interaction, only the current formulae are needed to describe the current state. A redefinition however occurs at a particular point in time. Making a redefinition is an act of agency — it is a *command*.

As a redefinition occurs at a particular point "in time"[5], a formula within a redefinition may meaningfully contain evaluation (which is denoted by surrounding an expression with '`| |`' vertical bars in the ADM script syntax). Slade [Sla90, §2.1.1] gives the following example of a redefinition:

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

resulting in the definition formula associated with `new_currency` becoming set to:

```
new_currency = 5.22 * amount
```

if the value of `exchange_rate` at the time of redefinition is `5.22`.

The ADM provides a way of partitioning state that is related to the description of an agent in an LSD account. Definitive variables are organised into named groups called *entities*. A *program store* P holds entity descriptions. Entities are instantiated from their description in P, whereupon their definitive variables are added to the definition store D, which holds definitive variables that are *owned* by currently instantiated entities. Several further *commands* are provided by the ADM in order to manage the descriptions of entities in P and to instantiate them in D. An example will be given at the end of the next section, which explains how the ADM may be 'programmed'.

The instantiation of entities in D and A from descriptions in P is analogous to instantiation of objects from classes in object-oriented programs (of which the first example was SIMULA 67 [BDMN79]), but the ADM does not provide the now-common full object model, with a class hierarchy and inheritance.

In principle, use of dependency in the ADM extends to output. Slade describes how this may be envisaged [Sla90, §2.3.2]:

> 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.

[5] More precisely, in a particular definitive state.

> 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*

I demonstrate a very direct form of this kind of 'definitive output' in §3.5.4, where the video hardware of the DAM machine is used to directly display internal definitive state.

2.1.3 'Programming' the ADM

The previous section described how the ADM provides definitive state and allows the modeller to make transitions using redefinitions. Using the facilities described so far, no change occurs to the definitive state unless the modeller makes a redefinition — hence, the ADM supports 1-agent modelling.

The ADM also implements automated state-change, or *automated agency*. In the ADM, each state-changing *action* is conceptually made autonomously by an entity when a particular enabling condition is detected in the state. The enabling condition for an action is described by a *guard*. ADM scripts are therefore similar to LSD accounts, in which a privilege for a certain action is expressed by the use of a guard. However, a guard in LSD describes only a necessary condition: when the guard is true, the LSD action *can* be performed. A guard in the ADM is closer to the spirit of [Dij76]: when the guard is true, the action *will* be performed.

In the ADM, an action is described using the following BNF[6].

action ::= *guard* → *command_list*
guard ::= *boolean_formula*
command_list ::= *command* (; *command*)*
command ::= *redefinition* | *instantiation* | *deletion*

[6]Note the use of the symbol → which does not appear on a conventional keyboard — the syntax given here is for the ADM, not an implementation.

```
entity one(first) {
    definition
        var1 = first,
        change1 = (var1  > var2)
    action
        change1 → var1 = |var1 - var2|,
        !change1 && !change2 → output = |var1|; delete one(first)
}

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

Listing 2.1: Two ADM entities that can compute GCD

Slade [Sla90, §2.1.3] gives the following examples of actions:

```
¬ can_start → choke = true;
(time == 2000) → switch = false; alarm = true;
```

Actions are mapped to entities in a similar manner to definitions. The entity descriptions in the program store P contain information about the potential actions performed by each entity. When the entity is instantiated from P, its actions are added to the *action store* A, which holds action descriptions that are owned by currently instantiated entities.

Listing 2.1 (taken from [Sla90, §3.7], with minor changes from `am` to ADM syntax) gives an example of an ADM script containing two entities that interact to eventually calculate the GCD of two integers (provided as parameters to the entities when they are instantiated) in a form of a "dance"[7]. Figure 2.3 gives the BNF that describes interaction with the ADM. I have constructed the figure by reference to [Sla90] (who gives a detailed description of the syntax accepted by his `am` implementation but no complete syntax for ADM scripts). The particular algebra used is not specified

[7] [BR92] gives details of a "folk-dance routine" for computing GCD.

in the ADM, and so the syntax for expressions is undefined, aside from the vertical bar '`| |`' syntax for evaluation. Figure 2.4 (which is based on [Run02, Figure 3-4, p.82]) illustrates the structures contained in the ADM during use.

statement ::= *command* `|` *query* `|` *entity*

command ::= *redefinition* `|` *instantiation* `|` *deletion*

redefinition ::= *definition* `|`
var_name `=` *formula_with_evaluation*
definition ::= *var_name* `=` *formula*

instantiation ::= *entity_name* `(` [*parameter_list*] `)`
deletion ::= `delete` *entity_name* `(` [*parameter_list*] `)`

query ::= `?` *var_name*

entity ::= `entity` *entity_name* `(` [*parameter_list_description*] `) {`
[`definition` *definition_list*]
[`action` *action_list*]
`}`

definition_list ::= *definition* (`,` *definition*)*
action_list ::= *action* (`,` *action*)*

action ::= *guard* → *command_list*
guard ::= *boolean_formula*
command_list ::= *command* (`;` *command*)*

var_name ::= *string*
entity_name ::= *string*

Notes:

1. The syntax of *formula*, *boolean_formula* and *formula_with_evaluation* are undefined, in order to allow for many possible type algebras. An exception to this is that the vertical bar '`| |`' syntax is used to denote the evaluation of a sub-expression in *formula_with_evaluation* only.
2. The syntax of *parameter_list* and *parameter_list_description* are defined, but are omitted from this particular description. Further description of the issues surrounding parameter lists is given in [Sla90, §2.3.1].

Figure 2.3: BNF describing syntax for the ADM

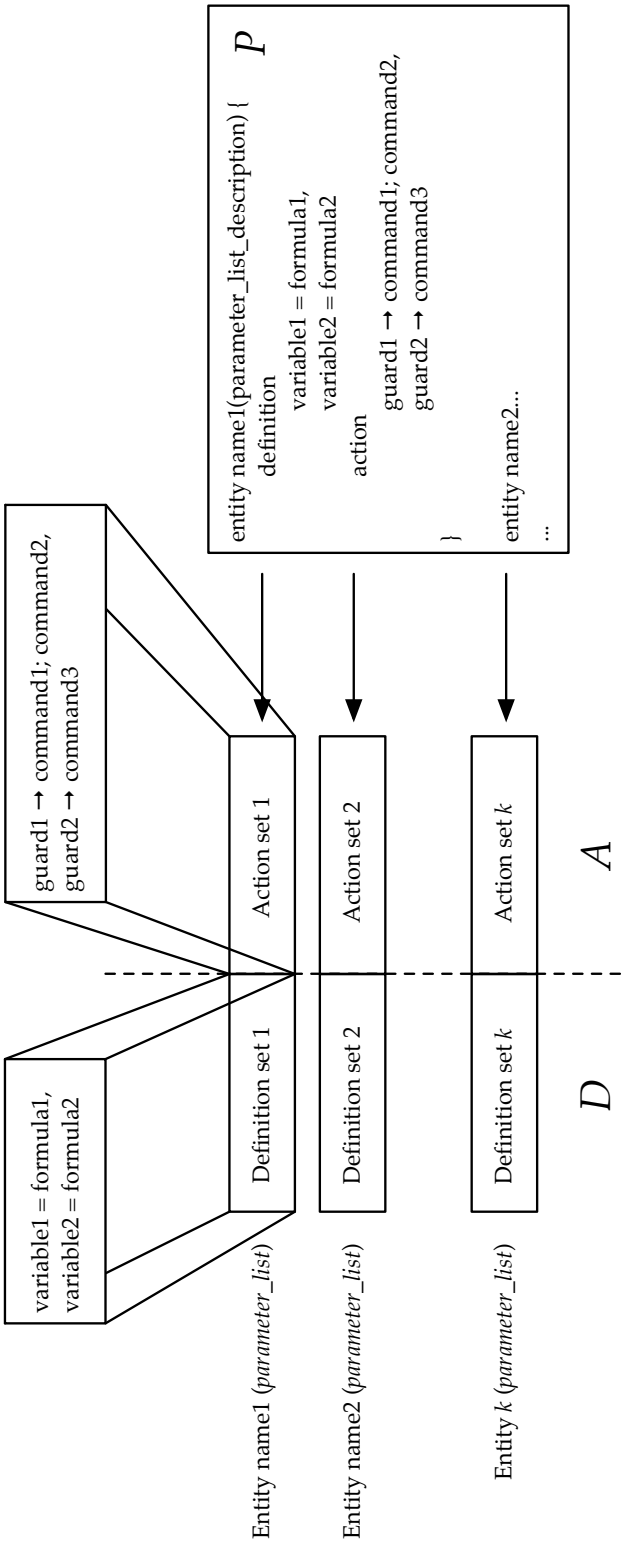

Figure 2.4: D, A and P in the ADM

2.1.4 From LSD accounts to ADM scripts

There are three significant differences between LSD accounts and ADM scripts. They reflect the fact that LSD was designed to model experience from multiple (possibly inconsistent) points of view, and the ADM was designed to model experience from the perspective of an external observer.

Perception or direct experience of values: There is no global state in LSD. In this respect, it is a "distributed computing model". An LSD agent with a `state` observable carries the *authentic* value of the observable. Agents can access the values of `oracle` observables only by *perceiving* them.

The independence of different perceptions is emphasised in Slade [Sla90, §5.3.1], who states "The authentic value and any perceived values of the same variable will be stored in private memory locations in different agents." The synchronisation between authentic and perceived values is not defined in LSD[8]. In the ADM, however, all state is global. In this respect, the ADM is a "shared-memory computing model". In EM terms, LSD accounts lack the external observer perspective which ADM assumes.

Guards as 'can' or 'will': In LSD, guards represent privileges for action. "In LSD a guard being true gives permission for the agent to act to change the state." [Sla90, §5.2]. In the ADM, however, "all guards which are true in an execution cycle have their associated command list executed: there is an obligation on the entity to execute its command list" [Sla90, §5.2].

Speed and synchronisation assumptions: In LSD, agents operate asynchronously. Their relative speeds of execution are undefined. "In general [agents] will operate at very different rates... Part of interpreting the specification as describing the behaviour of a system will involve deciding how fast agents operate relative to each other." [Sla90, §5.3.2]. LSD guards are also subject to undefined uncertainty: "There can be an arbitrary time interval between the selection of a true guard for execution and the sequential execution of its

[8] Except for in the case of derivates which refer to `state` observables of another agent, in which case the authentic value is used in "close synchronisation" — see [Sla90, §5.3.1 and §8.5].

Basic concept	LSD	ADM
agent	*agent*	*entity* (described in *P*)
observable	*state* of agent	*variable* owned by entity (in *D*)
	oracle of agent	variable *evaluated* by action owned by entity
	handle of agent	variable *redefined* by action owned by entity
dependency	*derivate* of agent: describes relationship between observables in the view of the agent	*definition* owned by entity (in *D*): describes relationship between variables in the view of the entity
agency	action of agent is permitted by a true guard *privilege*	*action* (in *A*) of entity is mandated by a true guard

Table 2.1: LSD and ADM terminology compared

command list" [Sla90, §5.2]. The operation of ADM entities, however, are synchronised by a global clock.

Beynon *et al* [BNOS90] and Slade [Sla90, §5.4] describe how an LSD account may be animated using an ADM script. An LSD account describes the interactions between agents in a concurrent system in terms of their privileges to perform actions. It identifies the characteristics of system behaviour that depend upon the interrelated capabilities and perceptions of its participating agents. A precise description of system behaviour requires additional information, as an LSD account can be given many behavioural interpretations, most of which are inappropriate. An LSD account can be translated into several ADM scripts, each associated with a different scenario for agent action. The terminologies used in the two notations differ in order to highlight these differences. The terms used are compared in Table 2.1.

2.2 Implementations of the ADM

2.2.1 Evaluation/storage implementation strategies for definitive systems

In [Sla90, §3.4], Slade notes that in the ADM, a redefinition does not necessarily imply an immediate evaluation and discusses when definitions should be evaluated, outlining some potential methods. My re-interpretation of the alternatives is shown below.

Strategy 1: evaluate at every use (storing only formulae)

Strategy 2: evaluate at every redefinition (storing formulae and values)

Strategy 3: a mix of 1 and 2: evaluate at use when a redefinition has previously out-dated the store (storing formulae, values and out-of-date flags).

I term these "evaluation/storage strategies" because they each demonstrate a progressive trade-off of storage for evaluation. The strategies are related to the 'data-driven' and 'demand-driven' classifications of computer architecture, described by Treleaven *et al* [TBH82]:

> In data-driven (eg, data-flow) computers the availability of operands triggers the execution of the operation to be performed on them, whereas in demand-driven (eg, reduction) computers the requirement for a result triggers the operation that will generate it.

The strategies are labelled in increasing order of implementation difficulty, assuming the implementation is constructed in a procedural language. Strategy 1 is relatively simple to implement in a procedural language as it corresponds to treating use of definitions as calls to functions[9]. Strategies 2 and 3 respectively use more storage[10] to reduce evaluation, adding more complexity.

Evaluation of just one definitive variable in any of these strategies can be a protracted task, as the variable may use many other nested definitions. The efficiency

[9] For example, the definition `a is b+c;` can be represented by the C function:
`int a() { return b() + c(); }`

[10] The three strategies are an abstraction involving some loss of detail. Although the `am` implementation uses strategy 1, it needs to store some state to implement evaluation in only an S state — see later.

of each strategy in terms of evaluation depends upon the use : redefinition ratio that is present in the script. In strategy 1, evaluation is initiated by use. This strategy is most efficient when redefinition is more frequent than use. In strategy 2, evaluation is initiated by redefinition. This strategy is most efficient when use is more frequent than redefinition. In strategy 3, the work is distributed somewhat across both use and redefinition: the out-of-date flags must be marked on redefinition, and evaluation must be performed at use if the value is found to be out-of-date. A sequence of redefinitions with no intermediate use therefore requires no evaluation, but the implementation must maintain the out-of-date flags. But note that when attempting to characterise the efficiency of this strategy, we are led to consider factors other than purely evaluation. More work and more storage is required to maintain the additional information needed for strategies 2 and 3.

The value of every guard expression in A is checked at the beginning of every transition in the ADM. The appropriate evaluation/storage strategy to use for guard expressions is therefore dependent on the ratio of guard evaluations to guard redefinitions.

The choice of evaluation/storage strategy is a fundamental initial decision taken when a definitive system is implemented and has wide-ranging effect. The three strategies therefore form the organising principle of this chapter and the following two chapters of this thesis. The implementation described briefly below is an example of strategy 1. The DAM machine, described in Chapter 3 is an example of strategy 2. Finally, EDEN, described in Chapter 4 is an example of strategy 3.

2.2.2 Existing implementations of the ADM

The ADM design has been concretely implemented more than once. However, at the time of writing, the implementations are few and all classed as prototypes. They fall into two categories:

- `am`
- `adm, adm2, adm3`

In this thesis, I will use ADM (uppercase) to denote the Abstract Definitive Machine concept outlined in the previous section, and lowercase implementation

names in Courier type to refer to implementations.

The first implementation to be constructed was named `am` and is presented in [Sla90, §3]. It is a "simulator, in the sense that the commands... are performed sequentially, rather than in parallel". The source code for `am` is written in the C language and the lex and yacc parser generators are used. The source is approximately 5000 lines of code (comments and blank lines included).

The second way that the ADM has been implemented involves translation from a script written in ADM-style syntax into Eden code. Y.P. Yung presents such an EDEN-based implementation named `adm` in [Yun93, §8.5], and later a second implementation giving more control over evaluation, named `adm2` [Yun96, §6.2]. P-H. Sun presents a third version named `adm3` [Sun99, §6.2.1] which attempts to simplify the syntax of the translated Eden output. Due to the translation, the implementation gains the benefit of Eden's graphical and interactive features, but the state-transition model becomes hard to analyse in isolation from EDEN's operational model (the subject of §4.3). These implementations therefore play a minor role in this chapter.

The full BNF grammar for the interaction language of the first implementation, `am`, is given in [Sla90, Appendix 4.2]. It is limited to integer values only but is otherwise quite like the syntax of the language 'C'. It extends the ADM BNF given in Figure 2.3 slightly, adding the ability to make a "procedural action" when a guard is true. The only available procedural action is a `print` command. This is a simple solution to the problem of output in `am` — the full ADM design (which is currently unimplemented) includes a scheme for input and output that is more consistent with a definitive model, described briefly in §2.1.2. The extension to the ADM BNF applicable in `am` is shown below.

action ::= *guard procedural_action* → *command_list*
procedural_action ::= `print("`*string*`")`

Appendix 2.A (p.91) gives an example of an `am` script and an example interaction, showing how the script can be loaded into the machine, entities instantiated, machine state can be queried, the machine can be invoked and redefinitions can be made.

It should be noted that, in concept, the ADM is substantially more 'embodied' than the prototype `am` implementation described here. For instance, Beynon maintains that definitive representations of screen state were within the scope of the

original ADM concept — in which case the choice of the epithet 'Abstract' for the ADM is somewhat misleading. In contrast, Slade's description [Sla90, §8.4.1] of an ADM interface using eight graphical windows under the heading "Extensions to the adm" is oriented towards giving the modeller more control over the execution of an abstract computational model.

The `am` interpreter uses evaluation/storage strategy 1. Slade justifies the choice with the following paragraph [Sla90, §3.4]. I have numbered the sentences for reference below:

> [1] There is reason to hope that many programs would consist of redefinitions of the same variable without intermediate evaluations being performed. [2] 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. [3] We suggest that this redundant evaluation is caused by the nature of procedural assignments, and can be avoided by the use of definitions. [4] 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.

Slade's first sentence is an assumption about ADM 'programs'. Sentence [2] in effect states that a definition maintainer using evaluation/storage strategy 2 (evaluate-at-redefinition) is inefficient if the assumption holds. Sentence [3] points out that a definition maintainer is free to delay evaluation until use. The last sentence states that this is the strategy used in the `am` implementation, which will be efficient if the assumption holds — if the assumption does not hold, then evaluation/storage strategy 2 may be more appropriate.

2.2.3 A hardware implementation?

Slade [Sla90, §3.4] suggests that the link in von Neumann architecture machines connecting store and CPU, that [Bac78] brands the "von Neumann bottleneck", poses a limitation in implementing "definitive programming", since in strategy 2,

> 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 transitions involved with this technique.

Slade then goes on to suggest a special purpose computer architecture more suited to the execution of "definitive programs". This text is paraphrased from [Sla90, §3.4]:

> If variables' values are to be available immediately to the CPU, there must be some mechanism to ensure that all dependent variables are updated after each redefinition. 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.
>
> 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 other computation (e.g. the evaluation of a guard once the values of all the variables in it have been received from store), the variables on the stack have their values updated by the store processor.
>
> 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.
>
> This machine may avoid the von Neumann bottleneck, since a write operation will only involve one write 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.

This thesis considers how dependency may be implemented on digital computers through study of the ADM, the DAM machine (Chapter 3) and EDEN (Chapter 4). The study gives us more understanding of how dependency might be implemented directly in hardware. In particular, we return to the idea of separating update (performed by Slade's "store processor") and change (performed by the various agents/entities) in §5.1.2.

2.3 Operational semantics

2.3.1 Operational semantics of LSD

Y.P. Yung [Yun93, p.144] (actually quoting [Sla90, §5.2]) describes how "the protocol [privilege] of an LSD agent should be interpreted":

1. All the guards are evaluated.
2. If at least one of the guards is true then a guarded action [with a true guard] is chosen arbitrarily, otherwise the guards are re-evaluated.
3. The action [command] list associated with the chosen guard is executed sequentially.
4. The procedure is repeated.

This algorithmic interpretation adds clarity in two particular semantic aspects described below, but it also contains a mistake.

The mistake in the description is the following: step 2 implies that if exactly one guard is true, the associated action *will* be chosen. However, if this LSD concept is intended to imply a privilege (which Y.P. Yung states in the following paragraph), then step 2 should imply that even though a guard is true, the LSD agent is *not* obliged to choose to perform the associated action. On this basis, a more appropriate specification for step 2 would be:

2. Select a true guard and perform the associated guarded action or return to step 1.

With or without the correction, two semantic aspects are made clear by the description. Firstly, the description makes no reference to context, apart from that pertaining to the guards. This is due to the distinction made in the interpretation of LSD between perceived and authentic values. Whether guards refer to authentic or perceived values is a question that requires addressing.

The second aspect that is made clear by the description is the extent to which an LSD agent can perform parallel action. Steps 2 and 3 restrict an agent so that it only performs one action at a time. Considered in programming terms, LSD is 'single-threaded'. This restriction stems in part from the original design intention of LSD to represent processes [BN87], but it is also centrally concerned with the role that dependency plays in the modeller's interpretation of state change. Although only one action is performed at a time by an LSD agent, the associated state-change may affect many observables indivisibly. One purpose of the LSD account is to separate the independent "centres of state change" [BRY90] that determine agency. To date, the application of LSD has focussed upon associating each strand of independent agency with a different agent. This is the simplest type of account. It may be

more realistic in some circumstances to recognise that the same agent is capable of parallel execution of more than one independent action.

Y.P. Yung considers the latter point in [Yun93, §8.4.2]. Some sub-sequences of commands in his LSD accounts are underlined. For example, Figure 2.2 contains the example:

privilege alighting[p] ∧ pos[p] == 0 ∧ door_open[d] ∧ ¬queuing[d]
→ pos[p] = 1; pat[p] = |at|; door_open[d] = false; pos[p] = 2;

In English, this example reads: if the passenger is alighting the carriage (getting off the train) and is standing in the doorway and the door is open and there is no queue of passengers in the doorway, then move onto the edge of the platform, take note of the identity of the station, shut the door and move onto the platform proper.

Y.P. Yung [Yun93, §8.4.1] notes of his underlined commands that they "should, in principle, be executed in parallel... [but] the current LSD notation has no provision for specifying synchronised actions [commands]." He suggests extending LSD with a new "parallel action [command] separator"[11]. The underlined commands could then be written:

(pos[p] = 1 // pat[p] = |at|);

Derivates are used in LSD to specify indivisibility in observation from the point of view of an agent. With this extension, the parallel command separator then specifies indivisibility in change made by an agent. Cartwright's Block Redefinition Algorithm in the DAM machine (see §3.1.2) and Y.W. Yung's `autocalc` mechanism in EDEN (see §4.3) have the same aim.

Another application for the parallel command separator is illustrated by considering the agency involved in playing a piano. If a scale is being performed, one note is played at a time. This might be represented using the following LSD fragment.

staccato_scale → c=down; c=up; d=down; d=up; e=down; e=up;

If a chord is played, several piano keys may be depressed "simultaneously" and then released. A scale played *legato* involves simultaneous release of one note and

[11] And goes on to give an example involving the parallel swapping of two observable values, which is not used here due to its formal 'programming' connotations.

depression of the next. Using the parallel command separator, these can be represented by the following LSD fragments:

> chord → (c=down // e=down // g=down); (c=up // e=up // g=up);
> legato_scale → c=down; (c=up // d=down); (d=up // e=down); e=up;

The proposed extension to LSD provides some limited scope for parallelism but does not begin to address the complex issues involved in attributing agency in general. For instance, should we regard the playing of a chord as three synchronised actions performed by independent fingers or an atomic action resulting from a single movement of the arm? A more general extension of LSD would allow the description of these two distinct understandings.

2.3.2 Invalid transitions in the ADM

The notion of the *invalid transition*, described by Slade in [Sla90, §2.1.3] is an important feature of the ADM:

> 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 which state such a transition would result in...
>
> A set of redefinitions which do not constitute an invalid transition can be performed in parallel.

(On this definition, a valid transition is context-independent. Note that there are also types of context-dependent error that can occur in a transition between definitive states: for example, the transition may introduce a graph cycle, although this may not be obvious from an examination of the set of redefinitions alone.)

Brinch Hansen [Han02b, p.22] defines a vocabulary which includes:

> **concurrent processes**, *processes* that overlap in time; concurrent processes are called **disjoint** if each of them only refers to **private data**; they are called **interacting** if they refer to **common data**.

The set of redefinitions in a *valid*[12] transition therefore exhibit the essential characteristic of a set of disjoint concurrent processes, albeit for only that single valid transition.

Brinch Hansen later goes on to state the importance of this concept ([Han02b, p.30], his emphasis):

> Hoare introduced the essential requirement that *a programming language must be secure* in the following sense: A language should enable its compiler and run-time system to detect as many cases as possible in which the language concepts break down and produce meaningless results.
>
> *For a parallel programming language the most important security measure is to check that processes access disjoint sets of variables only and do not interfere with each other in time-dependent ways.*

The ADM is intended to detect an invalid transition at run-time. An invalid transition may be an error fatal to the current automated agency in an ADM 'program'. However, [Sla90, §2.3.4] continues:

> It would be inappropriate to regard an invalid transition... as automatically indicating a flaw in the program being executed, since it may indeed be a faithful modelling of the real world... 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.

If the run set is valid, it can be executed in parallel and it is guaranteed that the resulting state is well defined. An invalid transition should therefore not be considered "just another sort of error": the important point is that a valid transition is interference-free. Valid transitions in the ADM therefore provide a powerful guarantee of valid state: since the machine will not proceed with sets of redefinitions that (when acting upon the current state) do not lead clearly to a single possible resultant state, the state is valid at all times. If an invalid transition halts automated agency in the ADM, the state is still valid and if the invalidity is resolved, automated progress can continue if desired.

[12]=Not invalid.

1. Check for cyclic dependency error in D.
2. For each guard in A:
   (a) Evaluate the guard.
   (b) If the guard is true, add the associated command list to the run set.
3. If run set is empty, halt execution.
4. Check run set for the various cases of interference error (an invalid transition).
5. Simulate parallel execution of all the command lists in the run set.
6. Go to 1.

Listing 2.2: Slade's ADM algorithm

2.3.3 What's in a transition?

A *transition* in the ADM is an important notion. Much of the semantics depend upon an exact definition of what constitutes a transition. In this subsection, I investigate the notion of the ADM transition in respect of the aspects listed below. The last aspect is a significant theme that runs through this thesis.

- Guard evaluation, Invalid transition
- Command sequencing
- Evaluation

Guard evaluation, invalid transition

With respect to guard evaluation and the identification of invalid transitions, a transition in the ADM is formed from a set of complete command lists. Listing 2.2 shows the algorithm that forms the basis for Slade's `am` implementation. In respect of guard evaluation, a transition is one execution of steps 1–5. Steps 2(b) and 5 make clear that within a single transition, a set of complete command lists are executed. I have compiled the listing from Slade's algorithm for the ADM execution cycle[13]

[13]It is unclear whether the algorithm presented in [Sla90, §3.6] describes 'the' ADM operational semantics in general or just a particular implementation: Slade states only that it is "an algorithm for the computation... [which] describes the execution involved in a single execution cycle...". The algorithm is certainly the basis for Slade's `am` implementation. In the absence of any other formal descriptions, here we take it to be a description of 'the' ADM.

[Sla90, §3.6], simplifying by omitting detailed error handling cases.

Guards are not re-evaluated during step 5. Slade makes this clear in the following text, taken from [Sla90, §2.1.3]:

> ... The definitive state reached after [previous redefinitions within the current command list] cannot be used for evaluating guards in — command lists are treated as a single transition, no matter how many redefinitions they involve.
> ...
> ... 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.

The computational model described in Listing 2.2 is thus superficially similar to the bulk-synchronous parallel (BSP) model (proposed by Valiant [Val90] in the same year as Slade's work and later developed by McColl *et al*), in that it repeatedly determines the work to be performed in the next 'super-step' (by evaluating guards) and then performs this work (the command lists) in parallel. However, the BSP model does not use definitive state.

Command sequencing

A family of redefinitions can be considered as a sequence or a set if they constitute a valid transition. From an abstract point of view, the ordering of the redefinitions does not matter as there is one well-defined resultant state.

However, the sequencing of commands (which as well as redefinitions, include instantiation and deletion of entities) can be considered significant. Conceptually, we would wish to model a person entering a modelled room as a simultaneous instantiation of the person entity and initialisation of the entity variables. However, in the ADM, the instantiation and initialisation are represented through the use of a sequence. This is the intended purpose of command lists in the ADM, as Slade describes in [Sla90, §2.2.2]:

> 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.

Ideally we would wish to construct more expressive definitive notations that allow instantiation and initialisation of blocks of state in one redefinition. However, in the

ADM, it seems that Slade found it necessary to implement sequential command lists, intending them to be used only for this limited purpose.

Evaluation

The final aspect in which the notion of the ADM transition is important is evaluation. My formulation of the relevant questions are as follows.

> Can the effect of intermediate commands in a command list be observed by:
>
> - the agent that instigated the change?
> - another automated agent?
> - the modeller? (who is also an agent)

Slade appears to be silent on these questions: the quotations given so far refer to evaluation of guards, and step 5 in Slade's ADM algorithm (Listing 2.2) states only "simulate parallel execution of all the command lists in the run set" and so is under-specified in this respect.

Beynon, however, informally describes the operation of the machine in many of the early sources. The representative quotation[14] below, with my emphasis added, is taken from [Bey90, §2].

> ... *Each action is a sequence of instructions*...
> ... On each machine cycle the guards associated with actions in A are evaluated in the context specified by the definitions in D. If there is no interference, those actions that are associated with true guards are then executed in parallel. *Evaluation required in a redefinition... is performed in the same context as guard evaluation.*

The two emphasised parts of the quotation can be deemed to be in conflict: if evaluation is always performed in the same state as guard evaluation, then later commands in a command list cannot observe the state changes made by commands earlier in the list, and therefore command lists can be considered to be sets as well as sequences. However, there is no conflict, as long as a qualification is made to the effect that actions are sequences in respect of instantiation and deletion only, as described above under 'command sequencing'.

[14] Similar descriptions appear in [Bey88, p.8], [BSY89, p.3] and [BNOS90, p.4].

To aid thought on the matters of transitions and evaluation, I propose the use of the terms 'major' and 'minor' transitions[15]. In the ADM, the 'transition' considered so far, which is made up of sets of entire command lists, can be named a *major transition*. Each individual command within a command list forms a *minor transition*. A major transition moves the state from an initial state S to a resultant state S'. Minor transitions may produce intermediate $S*$ states between S and S'. The question I posed under the heading of 'Evaluation' above then becomes: can intermediate $S*$ states be observed?

I use the major/minor, $S, S*, S'$ terminology later in the thesis when talking of dependency-as-agency. Regarding the ADM, the second emphasised statement in the quotation from Beynon above can be read as: evaluation in a minor transition observes the S state formed by the last major transition, not an intermediate $S*$ state formed by a minor transition.

Slade's `am` implementation follows this scheme, as can be seen from the experimental interaction shown in Listing 2.3 (which is slightly edited to reduce the size of the output). The entity `test` created in the listing contains two actions, which perform evaluation and change of the same variable in the two possible sequences:

```
b=|a|; a=2;
```

and

```
a=2; b=|a|;
```

Despite the differences in sequence ordering of the command lists, it can be seen that the value of variable `b` in the resultant S' state always takes the value of variable `a` as it was in the initial S state. Although the `am` implementation executes the commands in the command lists sequentially and in the provided order, the evaluations observe the initial S state. This is illustrated in Figure 2.5. As the intermediate $S*$ states (denoted in the figure by unfilled rectangles) are not observed, the sequential implementation can therefore be interpreted as acting in parallel, as shown at the bottom of the figure.

[15] A musical reader might imagine examples of these to be (c=down // e=down // g=down) and (c=down // e♭=down // g=down) respectively, but they are not defined in this way here — please read on.

```
$ /dcs/emp/empublic/bin/am-1.1
am-1.1> entity test() {
    definition
        a=0, b=0, g=0
    action
        g==1 print("g1") -> b=|a|; a=2; g=0,
        g==2 print("g2") -> a=2; b=|a|; g=0
}
am-1.1> test()
am-1.1> define a=42; define b=0;
am-1.1> define g=1;
am-1.1> l ds
Variable # 1: a = 42
Variable # 2: b = 0
Variable # 3: g = 1
am-1.1> start
 g1
am-1.1> l ds
Variable # 1: a = 2
Variable # 2: b = 42
Variable # 3: g = 0
am-1.1> define a=42; define b=0;
am-1.1> define g=2;
am-1.1> l ds
Variable # 1: a = 42
Variable # 2: b = 0
Variable # 3: g = 2
am-1.1> start
 g2
am-1.1> l ds
Variable # 1: a = 2
Variable # 2: b = 42
Variable # 3: g = 0
am-1.1>
```

(Note: User input is shown `like this`.)

Listing 2.3: An interaction with `am` demonstrating evaluation in S state

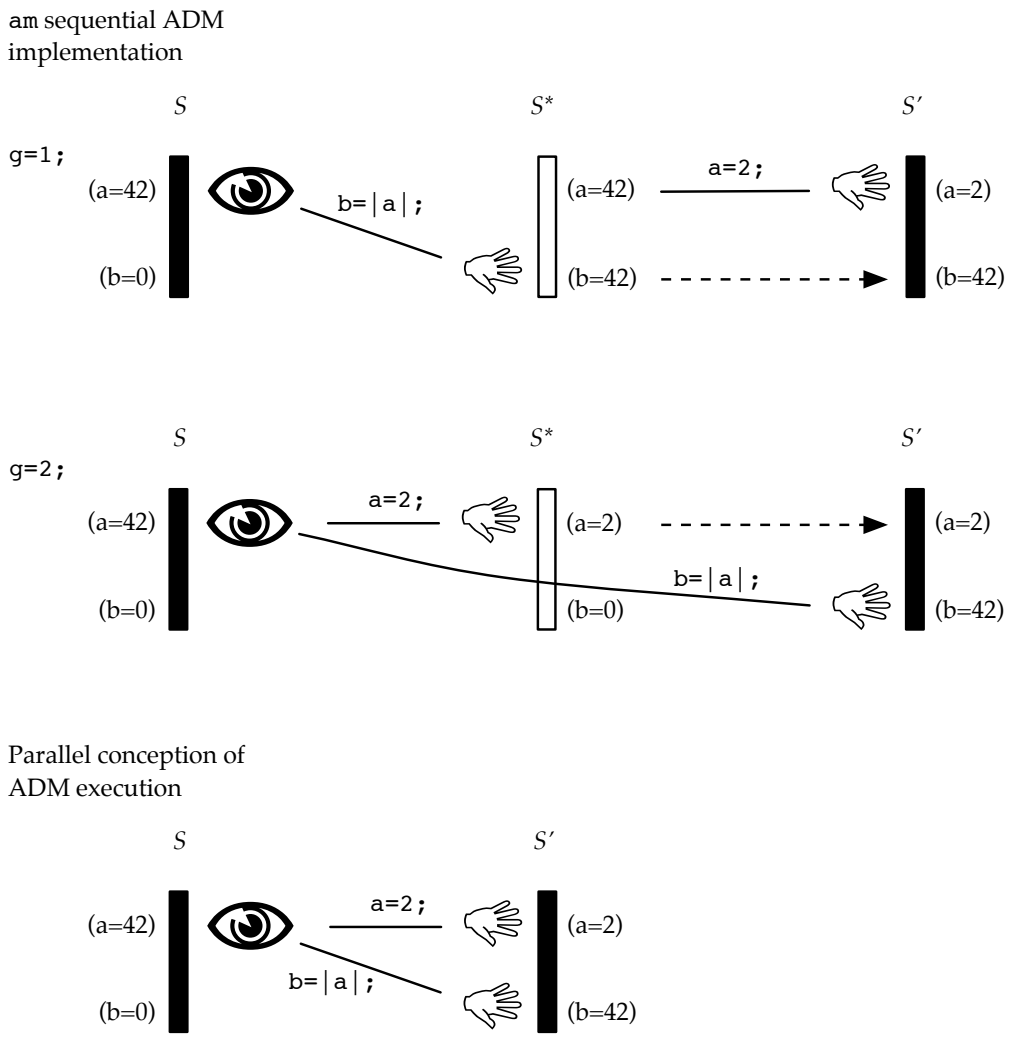

Figure 2.5: Sequential `am` implementation and parallel conception of the ADM

2.3.4 Divisible command lists and the 'Authentic' ADM (AADM)

Attempts have been made to use 'divisible' command lists in the ADM for simulation purposes, extending the use of command lists beyond the original purpose intended by Slade of supporting instantiation and deletion.

Y.P. Yung's railway passenger example (shown in Figure 2.2 on p.51 and considered earlier in terms of parallel action in §2.3.1) gives one example. Yung's LSD account includes the excerpt shown (slightly simplified) at the top of Figure 2.6. Yung translates this portion of the LSD account to the ADM script shown (again

agent passenger(···) {
    state pat[p], pos[p], alighting[p];
    oracle queuing[d], door_open[d];
    privilege alighting[p] ∧ pos[p]==0 ∧ door_open[d] ∧¬queuing[d]
        → pos[p]=1; pat[p]=|at|; door_open[d]=false; pos[p]=2;
    ...
}

```
entity passenger(...) {
    definition
        pat[_p] = ...,
        pos[_p] = ...,
        alighting[_p] = ...,
        state[_p] = 0,
        ...
    action
        alighting[_p] && pos[_p]==0 && door_open[_d] && !queuing[_d]
            print("Passenger ",_p," alighting on platform")
            -> pos[_p] = 1; state[_p] = 1; pat[_p] = |at|,
        state[_p]==1 && door_open[_d] && !queuing[_d]
            print("Passenger ",_p," closes door ",_d)
            -> door_open[_d] = false; state[_p] = 2,
        state[_p]==2
            print("Passenger ",_p," leaves the station")
            -> pos[_p] = 2; state[_p] = 0,
        ...
}
```

Figure 2.6: Excerpts from Y.P. Yung's railway passenger LSD account and ADM script

slightly simplified) at the bottom of Figure 2.6.

Notice how Yung introduces an additional state variable, named `state`, and splits the single LSD command list into multiple ADM actions, in part guarded by the additional variable `state`. This technique is necessary in order to implement divisible command lists in the ADM. LSD command lists conceptually execute sequentially and the intermediate states are visible to other agents. ADM command lists however execute conceptually in parallel and the intermediate states are not visible to other entities. Yung has determined, when translating from LSD to the ADM, that the LSD command list shown should be translated to the three ADM actions shown, the two underlined LSD commands being synchronised (as described

`am`		
	`a = b;`	*(re)definition*
	`a = |b|;`	*evaluation in S*
`adm`		
	`a is b;`	*(re)definition*
	`a = b;`	*evaluation in S*
`adm2`		
	`a is b;`	*(re)definition*
	`a = b;`	*evaluation in S∗*
	`a = |b|;`	*evaluation in S*

Figure 2.7: Syntax for redefinition and evaluation from three implementations of the ADM

in §2.3.1) and therefore being mapped to a single ADM action.

A separate, partial, attempt at using divisible command lists was made by Y.P. Yung in the second version of his ADM to Eden translator, `adm2`. Yung's first translator implementation of the ADM, named `adm`, was used by Emma Davis in 1995 for modelling interaction within a classroom [Dav95]. Her report states:

> The first difficulty I came across was that the lines of code within each action section were executed concurrently. While the action sections themselves should be executed concurrently, the code within them should be executed sequentially.

The operational semantics of Yung's `adm` therefore initially matched that of Slade's `am`. Yung solved Davis's problem by introducing a second kind of evaluation distinguished by a different syntax, described in [Yun96, §6.1] and produced a new translator, named `adm2`. The relevant parts of the syntax of the three implementations is shown in Figure 2.7. Yung's `adm` implementations deviate from the syntax given earlier in Figure 2.3 (p.56) as they use the `=` operator to denote evaluation and `is` to denote (re)definition, in order to ease translation to the Eden language. More important here are the varied operational interpretations. Figure 2.7 shows that `adm2` offers two types of evaluation.

Allowing evaluation in an intermediate $S*$ reveals the many possible command interleavings that are abstracted away by the ADM execution as described in the previous section. Yung [Yun96, §6.1] warns that "in a concurrent execution environment assigning variables in the context of execution, in principle, is dangerous; there is no guarantee what value you have assigned". Use of such evaluation in the example shown in Figure 2.5 could lead to the variable `b` in the resultant state taking the value of 42 or 2, depending upon the particular interleaving used by the implementation.

Slade's description of the ADM algorithm (Listing 2.2, p.69) and his associated observation to that effect that "command lists are treated as a single transition, no matter how many definitions they involve" indicate that he identified the ADM with a machine that executes parallel actions in an indivisible fashion. Yung's railway animation demonstrates that such an interpretation of ADM execution is consistent with the animation of an LSD account, subject to grouping the sequence of commands in the LSD agents into indivisible segments, and 'programming' the ADM to execute these accordingly. This approach to simulation is what Slade would have recognised as 'using the ADM to animate an LSD account'.

The correspondence between LSD and the ADM highlighted in Table 2.1 (p.59) commends a conceptually very different interpretation[16] of 'using the ADM to animate an LSD account'. In this approach to LSD animation, each LSD agent corresponds to an ADM entity and contributes a set of actions for which the command lists are to be conceived as sequences of atomic commands. For the purposes of simulating behaviour on the basis of an LSD account — bearing in mind the status of LSD actions as privileges rather than obligations for action — a mode of execution quite unlike that described by Slade in the ADM algorithm of Listing 2.2 is appropriate. I have specified this alternative "mode of execution" in Listing 2.4, which is based on Beynon's informal accounts (e.g. [BACY94a, BACY94b]) of how the ADM was intended to be used to give operational meanings to an LSD account.

[16] In Slade's terms, this would be seen as a misinterpretation, since it presumes that command lists that are divisible, and comprise sequences of atomic commands.

Listing 2.4 documents (seemingly for the first time in the EM literature) the initial conception of a framework for LSD animation by Beynon and Slade that led to the identification of the ADM. In Slade's account of this research in [Sla90], the term 'ADM' refers to the 'computational engine' that is used to implement the parallel execution of indivisible commands specified at Step 12 in Listing 2.4. Unfortunately, in writing about the use of the ADM in LSD animation, Beynon has consistently (e.g. [BACY94a, BACY94b]) referred to the ADM as if its authentic mode of execution was as I have detailed it in Listing 2.4. This usage of the term ADM in connection with divisible command lists differs radically from the conception of the ADM introduced by Slade, where command lists are understood to be indivisible. On this basis, I propose that Beynon's use of the term 'ADM' to refer to execution with divisible command lists be deprecated, and that the term 'Authentic' ADM (AADM) be adopted where the mode of execution specified in Listing 2.4 is presumed. It is to the AADM concept that Rungrattanaubol implicitly refers in the 'Abstract Definitive Modelling framework', "a conceptual framework for multi-agent construal" [Run02, p.83], introduced in her doctoral thesis [Run02, §3].

The AADM algorithm in Listing 2.4 combines a 'reliable' part for evaluating functional dependencies (a 'machine' component) and an 'unreliable' part for simulating agency (a 'human' component). Entities in a typical execution of the AADM are derived directly from state observables of agents in an LSD account (*cf.* Table 2.1, p.59). The AADM then serves as "an instrument for interpreting[17] LSD".

A transition in the AADM algorithm is formed from a set of single commands selected from command lists. Each command list is executed sequentially, as in the ADM. However, each command is itself atomic and the resulting intermediate $S*$ states (between the commands in a command list) are observable to other agents. The commands within the set of active command lists are interleaved non-deterministically.

In the context of Listing 2.4, a command is intended to denote an "atomic action" on the part of an agent. This would normally be represented by a redefinition or an entity invocation/deletion. In some circumstances, it may be appropriate for a command to take a composite form. For instance, as discussed in §2.3.1, a

[17] In both the human and machine senses.

In each step:

1. (The state is now S.)
2. For each action a:
    3. If the action a is currently executing and there is no command from a already in the run set (pending execution):
        4. Add the next command in action a to the run set.
    5. Else:
        6. Evaluate guard of a in state S.
        7. If guard of a is true:
            8. Add the first command in action a to the run set.
9. Check the run set for an invalid transition.
10. If the transition is invalid,
    11. Stop and ask the modeller to resolve the conflict before proceeding.
12. Select a subset of the commands from the run set and execute these, conceptually in parallel, making the transition to state S'.
13. (The state is now S'.)
14. Go to 1.

Notes on step 12:

- Selection of the subset of commands can be determined non-deterministically by the algorithm or determined by the 'super-agent' modeller.
- Due to the guarantee given by the invalid transition check, the run set contains no interference between commands.
- Commands can therefore be performed sequentially *or* in parallel.
- If commands are performed sequentially, the state will move through intermediate $S*$ states before it arrives at S'. Evaluations can be performed in the S or $S*$ states without any difference to the result as there is no interference between commands.

Listing 2.4: A proposed algorithm for 'Authentic' ADM (AADM) execution (Ward, after Beynon and Slade)

single agent may be deemed to perform several actions in parallel ("the pianist plays a chord"). An entity invocation may also involve the introduction of many redefinitions in parallel ("Meurig enters the office"). Both of these scenarios can be regarded as instances of "block redefinition". Composite sequences of instructions may also be required in implementing commands. Slade's use in the ADM of a sequential but atomic "command list" for instantiation and initialisation of an entity (*cf.* §2.3.3) is the prototypical example. More generally, depending on the nature of the definitive notations at the disposal of the modeller, it may not be possible to represent the transformation of a complex geometric object by a single redefinition. For instance, relocating a 3-dimensional object may presume parallel redefinition of many coordinates. Even where definitive notations to address such a requirement exist, the indivisibility of redefinition must be guaranteed in the implementation. More complex issues of atomicity are raised by transitions that involve what previous authors have described as "Higher Order Definition" (HOD). Topics relating to the above concerns are addressed throughout the thesis — see for example §3.1.2 on block redefinition and §4.3.7 and §5.3 on HODs. They are associated with the agenda of "Implementing Dependency on Digital Computers".

The use of the word 'select' at step 12 reflects the idea that the non-determinism is potentially under the control of the modeller as "super-agent". Non-determinism and super-agency are concepts discussed by Slade[18] in [Sla90, §8.4] under the heading "Extensions to the adm", but are absent from the detailed descriptions of the ADM itself, as the above subsections have discussed. In discussing how LSD accounts and ADM artefacts are related in EM, Beynon frequently alludes implicitly to three different modes of execution of the ADM. These correspond to three strategies for implementing the selection of commands to be performed at step 12:

- Manual selection of commands by the modeller;
- Automated selection of specific commands following some empirically plausible pattern (for example, using probabilistic techniques);
- Automating the selection of *all* commands other than those that are intended to represent actions on the part of human agents.

[18] Calling them asynchronous execution and 'superuser' respectively.

Which of these strategies is appropriate depends in general on the maturity of the modeller's understanding of agency as it develops during the modelling activity (*cf.* the reference to View 1, View 2 and View 3 agents in §2.1.1).

The AADM will not be considered further here, as it is most closely associated with the semantics of automated agency, and the primary concern of this thesis is the implementation of dependency.

2.4 The ADM and UNITY

UNITY ([CM88], on which much of the following description is based) is a theory of programming, comprising:

- a method for specification of programs;
- a method of reasoning about specifications;
- a method of developing programs with a proof that they meet their specification;
- a method of transforming programs to achieve high efficiency on the machines available for their execution.

UNITY generalises the theory of sequential programming [Dij76] for use with distributed and concurrent algorithms. Although some notational syntax is given in [CM88] for UNITY, UNITY is not considered a programming language by the authors — the notations are introduced in order to illustrate their computational model and proof system. UNITY stands for Unbounded Nondeterministic Iterative Transformations, describing a concurrent computational model that can be mapped to other computation models, including sequential ones.

UNITY programs consist of a declaration of variables, a specification of their initial values and a set of *multiple-assignment* statements. Figure 2.8 shows a selection from the BNF description of the UNITY notation. Syntactic units enclosed between { and } may be instantiated zero or more times.

program	::=	**Program**		*program-name*
		declare		*declare-section*
		always		*always-section*
		initially		*initially-section*
		assign		*assign-section*
		end		
assign-section	::=	*statement-list*		
statement-list	::=	*statement* { [] *statement* }		
statement	::=	*assignment-statement* | *quantified-statement-list*		
assignment-statement	::=	*assignment-component* { ‖ *assignment-component* }		
assignment-component	::=	*enumerated-assignment* | *quantified-assignment*		
enumerated-assignment	::=	*variable-list* := *expr-list*		
variable-list	::=	*variable* {, *variable*}		
expr-list	::=	*simple-expr-list* | *conditional-expr-list*		
simple-expr-list	::=	*expr* {, *expr*}		
conditional-expr-list	::=	*simple-expr-list*	if	*boolean-expr*
		{ ∼ *simple-expr-list*	if	*boolean-expr*}

Figure 2.8: Partial BNF for UNITY (summarised from [CM88, §2])

Some examples of *enumerated-assignments* are given in [CM88, §2.3.2]:

1. Exchange x, y.

   $$x, y := y, x$$

2. Set x to the absolute value of y.

   $$x := y \text{ if } y \geq 0 \sim -y \text{ if } y \leq 0$$

3. Add $A[i]$ into sum and increment i, provided i is less than N.

   $$sum, i := sum + A[i], i + 1 \text{ if } i < N$$

4. Assign the smaller of $A[i]$ and $B[j]$ to $C[k]$ and increment k by 1; also increment i if $A[i] \leq B[j]$ and increment j if $B[j] \leq A[i]$.

   $$\begin{array}{rrcll} & C[k] & := & \min(A[i], B[j]) & \\ \| & k & := & k+1 & \\ \| & i & := & i+1 & \text{if } A[i] \leq B[j] \\ \| & j & := & j+1 & \text{if } B[j] \leq A[i] \end{array}$$

   or

   $$\begin{array}{llllll} C[k] & , i & , j & , k & := & \\ A[i] & , i+1 & , j & , k+1 & & \text{if } A[i] < B[j] \sim \\ B[j] & , i & , j+1 & , k+1 & & \text{if } A[i] > B[j] \sim \\ A[i] & , i+1 & , j+1 & , k+1 & & \text{if } A[i] = B[j] \end{array}$$

An *assignment-component* can also be a *quantified-assignment*, allowing a finite number of instances of assignment to be created from the one specification. The brackets $\langle$ and $\rangle$ delineate quantification scope.

quantification	::=	*variable-list* : *boolean-expr* ::
quantified-statement-list	::=	$\langle$ [] *quantification statement-list* $\rangle$
quantified-assignment	::=	$\langle$ $\|\|$ *quantification assignment-statement* $\rangle$

Some examples of *quantified-assignment*s are given in [CM88, §2.3.3]:

Given arrays $A[0..N]$ and $B[0..N]$ of integers, assign $\max(A[i], B[i])$ to $A[i]$, for all i, $0 \leq i \leq N$.

$$\langle \| \; i : 0 \leq i \leq N :: A[i] := \max(A[i], B[i]) \rangle$$

or

$$\langle \| \; i : 0 \leq i \leq N :: A[i] := B[i] \text{ if } A[i] < B[i] \rangle$$

or

$$\langle \| \; i : 0 \leq i \leq N \wedge A[i] < B[i] :: A[i] := B[i] \rangle$$

Program execution in the UNITY model starts from any state satisfying the initial condition and goes on forever. In each step of execution, some assignment statement is selected non-deterministically and executed. The non-deterministic selection is constrained by the "fairness rule": every statement is selected infinitely often.

Sequential control flow is therefore absent in the UNITY programming model. The passage below [CM88, §1.2.2] explains why.

> The notion of sequential control flow is pervasive in computing. Turing Machines and von Neumann computers are examples of sequential devices. Flow charts and early programming languages were based on the sequential flow of control. Structured programming retained sequential control flow and advocated problem decomposition based on the sequencing of tasks. The prominence of sequential control flow is partly due to historical reasons. Early computing devices and programs were understood by simulating their executions sequentially. Many of the things we use daily, such as recipes and instructions for filling out forms, are sequential; this may have influenced programming languages and the abstractions used in program design.
>
> The introduction of co-routines was an indication that some programs are better understood through abstractions unrelated to control flow. A program structured as a set of processes is a further refinement: it admits multiple sequential flows of control. However, processes are viewed as *sequential* entities — note the titles of two classic papers in this area, "Cooperating Sequential Processes" in Dijkstra [Dij68] and "Communicating Sequential Processes" in Hoare [Hoa78]. This suggests that sequential programming is the norm, and parallelism, the exception.
>
> Control flow is not a unifying concept. Programs for different architectures employ different forms of control flow. Program design at early stages should not be based on considerations of control flow, which is a later concern. It is easier to restrict the flow of control in a program having few restrictions than to remove unnecessary restrictions from a program having too many.
>
> The issue of control flow has clouded several other issues. Let us review one. Modularity is generally accepted as a good thing. What is a module? It implements a set of related concerns, it has a clean, narrow interface, and the states of a system when control flows into and out of the module are specified succinctly. Now a clean, narrow interface is one issue and control flow into and out of a module is another. Why not separate them? In our program model, we retain the concept of a module as a part of a program that implements a set of related concerns. Yet we have no notion of control flow into and out of a module. Divorcing control flow from module construction results in an unconventional view of modules and programming — though a useful one, we believe, for the development of parallel programs.

Control flow is however specified in UNITY: not in the programming model, but in the mapping of a UNITY program to an architecture. A mapping specifies how to partition the set of statements of a program amongst processors, how to

partition the set of variables among memories and channels, and the control flow for each processor. The mappings to synchronous architectures considered in [CM88, §4.2.3] are "particularly simple... In the mapping employed in this book, exactly one statement of the program is executed at a time, regardless of the number of processors available".

There are many similarities between the UNITY model and the ADM. In an intriguing parallel with the motivation behind the work presented in this thesis, "some of the initial motivation for UNITY came from difficulties encountered in using spreadsheets, a notation that has not received much attention from the computing-sciences community" [CM88, p.19].

Conditional assignments in UNITY correspond closely to guarded command lists in ADM scripts. Compare the ADM script fragment:

$$y \geq 0 \rightarrow x = y,$$

$$y \leq 0 \rightarrow x = -y$$

to the UNITY (from [CM88, p.25])

$$x := y \text{ if } y \geq 0 \sim -y \text{ if } y \leq 0$$

for example.

Chandy and Misra [CM88, §22.7.3] compare UNITY's conditional assignments to guarded commands as defined in [Dij76]. Aside from the syntactical differences shown in the example above (mainly relating to whether the conditions/guards or actions are emphasised), there is also a semantic distinction, which the following passage explains:

> The semantic differences between guarded commands and UNITY have to do with fairness. In the guarded command theory, a statement is selected for execution only if its guard is *true* (or "enabled"). If an arbitrary statement is selected for execution, it is possible that a statement whose guard is *false* is chosen forever — because there is no fairness constraint — and then there is no progress of computation in this case. Therefore the notion of a guard is crucial; only the statements with enabled guards are eligible for execution.
>
> In UNITY, the fairness rule obviates the need for guards. A statement whose execution does not change the program state may be selected for execution; it can, however, be executed only a finite number of times. Therefore a statement whose execution changes program state — if such a statement exists — is selected eventually for execution. This is how progress of computation is guaranteed in UNITY. There is no notion of guards or enabling, and this has simplified the proof theory.

The motivation for the fairness rule in UNITY is that of simplifying reasoning about progress properties [CM88, p.489]. The radical form of non-determinacy conceived in the AADM is also a departure from the traditional operational interpretation of guarded commands. It is motivated by a desire to match simulation behaviour to behaviour as it is observed in the referent.

UNITY programs can contain an "always-section" which is similar to the definitions section in an ADM script. The always-section in UNITY "is used to define certain program variables as functions of other variables" [CM88, §2.7]. The syntax of the always-section follows that of the "initially-section", which follows that of the "assign-section", except that in assignments '`:=`' is replaced by '`=`'. A variable appearing on the left side of an equation in the always-section is called a "transparent" variable.

> A transparent variable is a function of nontransparent variables and hence does not appear on the left side of any initialization or assignment, though it may appear on the right. A transparent variable may also appear on the right side of an equation.

To ensure that each transparent variable is a well-defined function of nontransparent variables, UNITY places restrictions on transparent variables. The restrictions are the same as for variables in the initially-section. The equations in the initially-section and transparent variables should not be circular. The set of equations must be "proper", defined as [CM88, §2.5]:

1. a variable appears at most once on the left side of an equation,
2. there exists an ordering of the equations such that any variable in a quantification is either a bound variable or a variable that appears on the left side of an equation earlier in the ordering (ensuring that the program can be "compiled"), and
3. there exists an ordering of all equations after quantified equations have been expanded such that any variable appearing on the right side of an equation, or in a subscript, appears on the left side of an equation earlier in the ordering (ensuring that the initial values are well defined).

An example given ([CM88, §2.7]) for a transparent variable is

$$ne = nf + nm$$

where *nf*, *nm*, *ne* denote the number of female employees, male employees and employees respectively.

The following passage ([CM88, §2.7]) motivates the always-section:

> ... First, it is simpler to reason with an always-section because it defines a set of invariants of the program... These invariants are in a particularly nice form, a set of equations. If a program contains only transparent variables, it can be regarded purely as a set of equations, and it is usually easier to understand such programs. Second, it is convenient to regard a transparent variable merely as a macro whose definition can be substituted for its occurrence anywhere in the program. The term transparent comes from this property of *referential transparency*. Third, efficient implementations of transparent variables are possible: Evaluation of a transparent variable can be deferred until it is accessed or until some of the variables in its definition change value.

Transparent variables in UNITY are thus similar to definitions in the ADM. However, the UNITY always-section cannot be changed at run-time: the always-section specifies invariants, or fixed constraints. This assists with a proof of correctness. The ADM in comparison could be thought of as allowing a sequence of "run-times", the sequence being formed through modification of definitions. The "transparent" nature of the definitions makes analysis of the static state possible: definitions provide meaningful state.

Finally, Chandy and Misra [CM88, Epilog] propose what might be called a methodology for developing UNITY programs.

> The change in a programmer's primary concern, from a specific machine — i.e., one in Taylor Hall — to a generic machine — i.e., a FORTRAN machine — was the first in a series of steps towards increasing abstraction... But then computer architects began introducing strange machines. They demolished the comforting cocoon of conformity.
>
> The UNITY response to the challenge of novel machines is to propose yet another answer to the question What is programming? Our answer is a logical progression to the answers given by our programming forebears. Once again we generalize our view of programming.
>
> Programming is the art of making a series of stepwise refinements of specifications. We begin with a large space of potential solutions, each refinement rules out some solutions, and we end with a small number of "good" concrete solutions. We expect that the decisions we make early in our designs are appropriate for *all* architectures. As design proceeds and the target architectures are defined more narrowly, our decisions are more appropriate for smaller sets of architectures. This approach is not unlike that made by programmers who, when opting for one PASCAL data structure rather than another, expect their decisions to be appropriate for *all* machines on which their programs run. In the final stages of design, programmers may code a few subroutines in assembly language to exploit features of a target machine. But programmers know that it is not cost effective to *begin* design by programming in the assembly language of all machines on which their programs may be required to execute.
>
> The core of programming is not concerned with a specific architecture, a specific operating system, or a specific programming language any more than it is

> concerned with the machine in the basement. The core of programming is a theory that allows programmers to make a series of design decisions. This book has attempted to present such a theory.

The above discussion of UNITY is of interest because of several points of contact with the concepts behind the ADM and its use in animating LSD. The ADM that Slade identifies can be seen as an "abstract computational model" that could in principle be the basis for a formal specification of behaviour in much the same spirit as UNITY. However, the distinctive perspective that is characteristic of EM is only apparent when the AADM is considered. The simulation activity specified in Listing 2.4 (p.79), which combines manual and automated interaction, is a more closely integrated concrete and situated activity than is normally associated with "a refinement methodology". The perception of indivisibility and the engineering of artefacts through experimental interaction play an essential role in this activity. For more details, and an elaboration of the theme, the interested reader may consult Rungrattanaubol [Run02]. In the context of this thesis, the most significant implication is that the implementation of dependency and agency in EM tools cannot be treated as an abstract programming exercise. The way in which state is perceived and engineered is crucial to the successful implementation of AADM simulation.

2.5 Background/Sources

The Abstract Definitive Machine has been written about in many papers and theses.

The papers can be categorised into two approximate themes: parallel programming and design and modelling. Parallel programming was an early theme for publications containing material on the ADM. The first publication is [Bey88], which gives the first treatment of a block-moving example, followed by [BSY89] which describes a simulation of a systolic array algorithm, [Bey90] which gives the fullest treatment of the block-moving example and [BNOS90] which discusses the relationship of LSD accounts and the ADM in the context of modelling a telephone exchange.

In the design and modelling theme papers category, [Bey89c] was the first journal publication to present the block-moving example (an example which straddles both thematic categories) as an ADM 'program' along with a graphical representation

in DoNaLD. [Bey89a] is concerned with definitive programming in design, [BRY90] presents an LSD model of a cat flap, [BY92] considers modelling and simulation of activity at a railway station as an LSD account and an ADM 'program', [BC93] envisages some extensions for specifying ADM entities and finally [BACY94a] contains a figure representing the ADM machine model. The ADM also gets brief mention in many other papers, including [BY90, BACY94b, BACY94c, Bey94, BY94, BC95, Bey97, Bey99].

The most detailed source about the ADM ideas is Slade's MSc thesis [Sla90], which is quoted extensively in this chapter. It contains the original treatment of the block-moving example and the telephone exchange model and also describes the first implementation `am`.

Y.P. Yung's PhD thesis [Yun93], [Yun96] (a post-doctoral report) and P-H. Sun's PhD thesis [Sun99] present ADM 'program' to Eden translators named `adm`, `adm2` and `adm3`, which are not the main focus of this thesis chapter. Additionally, [Yun93, §2.1] describes the Definitive State Transition (DST) model, similar to the ADM machine model. Rungrattanaubol [Run02] reviews the area, somewhat confusingly reusing (§3.1) the upper case acronym ADM for Abstract Definitive Modelling (lower case adm is the Abstract Definitive Machine). Heron [Her02] shows how to convert an ADM 'program' into a JaM2 script [Car99]. Wong [Won03, §2.2] describes the Definitive Modelling Framework (DMF), which is also similar to the ADM model.

A reader of the sources needs to be aware that sometimes the source is describing LSD, sometimes ADM the machine model, or sometimes an implementation of the ADM, which might itself be a full machine implementation or a translator into some other language. Often, the focus is not explicitly declared, and so clues must be obtained from the context. Table 2.1 (p.59) assists in distinguishing LSD and ADM terminology. Distinguishing discussion of the ADM from that about an implementation is sometimes more problematic. Slade [Sla90] uses mathematical symbols (e.g. $\wedge$) when describing the ADM machine model and ADM 'programs' in the abstract but keyboard symbols (e.g. `&&`) when describing the `am` implementation.

My scholarship on the ADM literature has exposed a significant misunderstanding that has had an unhelpful influence over the comprehension and practical development of the ADM since Slade's first implementation of `am`. It is apparent in

retrospect that the embryonic nature of the `am` implementation and confusion surrounding the authentic mode of execution of the ADM has inhibited the development of good tools to support EM development of concurrent systems. In particular, none of the existing EM tools combines manual and automated interaction, and divisible and indivisible execution of command lists, in a way that does justice to Beynon and Slade's original conception for LSD animation as I have identified it in this chapter. It is also clear that a successful implementation of a tool to support this conception will involve a more radical kind of revision of the `am` than has been envisaged to date; unlike `adm`, `adm2` and `adm3`, which are essentially rooted in automating atomic transitions, it will need to take full account of the partially-automated mechanism I have documented in Listing 2.4 (p.79). Some of the key technical issues to be addressed in developing such an implementation form the subject of the rest of my thesis.

2.A Using the am implementation

Listing 2.5 is a script for am based on a version by Y.P. Yung [BRY90] that models a cat, a man and a cat flap with a four-way lock. Notice, for example, that the man entity will set the cat flap lock to position 0 approximately once every 60 transitions, as long as the flap at angle 0 and the lock is not already at position 0.

```
entity flap() {

definition
    lflap = 5,                      # the length of the flap
    angle = 0,                      # inclination of the flap
    switch = true,                  # whether the electronic lock is
                                    # operating
    fourWayLock = 0,                # 1: cat can go out only, 2: in only
                                    # 0: no restriction, 4: no access
    Radius = 10,                    # electronic lock detector range
    elecLock = (pos > Radius || pos < -Radius) && switch,
    canPushOut = angle != 0 || (!elecLock && (fourWayLock == 0 ||
                    fourWayLock == 1)),
    canPushIn = angle != 0 || (!elecLock && (fourWayLock == 0 ||
                    fourWayLock == 2))

action
    pushOut && canPushOut
print("Angle becomes ", angle+1)
        -> angle = |angle| + 1,
    pushIn && canPushIn
print("Angle becomes ", angle-1)
        -> angle = |angle| - 1,
    !pushOut && !pushIn && angle > 0
print("Angle becomes ", angle-1)
        -> angle = |angle| - 1,
    !pushOut && !pushIn && angle < 0
print("Angle becomes ", angle+1)
        -> angle = |angle| + 1

}
```

```
entity man() {

definition
    actnow = 0

action
    actnow == 1 && switch == false
print("Switch on")
        -> switch = true,
    actnow == 11 && switch == true
print("Switch off")
        -> switch = false,
    actnow == 21 && angle == 0 && fourWayLock != 0
print("Four way lock is set to 0")
        -> fourWayLock = 0,
    actnow == 31 && angle == 0 && fourWayLock != 1
print("Four way lock is set to 1")
        -> fourWayLock = 1,
    actnow == 41 && angle == 0 && fourWayLock != 2
print("Four way lock is set to 2")
        -> fourWayLock = 2,
    actnow == 51 && angle == 0 && fourWayLock != 3
print("Four way lock is set to 3")
        -> fourWayLock = 3,
    true -> actnow = |rand(60)|

}

entity cat() {

definition
    height = 2,                     # height of the cat
    pos = 2,                        # >0: outside the house, <0: inside
    obstruct = 0,
    intention = -1,                 # 1: going out, -1 coming in,
                                    # 0 stay put
    pushOut = intention > 0 && obstruct,
    pushIn = intention < 0 && obstruct

action
    intention > 0 && !obstruct
print("Cat moves outward")
        -> pos = |pos| + 1,
    intention < 0 && !obstruct
print("Cat moves inward")
        -> pos = |pos| - 1
}
```

Listing 2.5: Y.P. Yung's am cat flap script

Interaction with am in use is through a text interface only. Listing 2.6 shows an interaction with am[19] that uses the script in Listing 2.5. User input is shown like this.

```
$ cd ~empublic/projects/catflapYung1994
$ cat flap-noinstantiate.am - | /dcs/emp/empublic/bin/am-1.1
am-1.1> compiling flap()
am-1.1> compiling man()
am-1.1> compiling cat()
am-1.1> l en                          # list entity descriptions (P)

        ENTITY LIST
        ***********
entity flap() {     (0 parameters)
...

entity cat() {      (0 parameters)
DEFINITION
    height = 2,
    pos = 2,
    obstruct = 0,
    intention = -1,
    pushOut = intention>0&&obstruct,
    pushIn = intention<0&&obstruct
ACTION
    intention>0&&!obstruct print("Cat moves outward") ->
        pos = |pos|+1,

    intention<0&&!obstruct print("Cat moves inward") ->
        pos = |pos|-1
}
0 instances
        END OF ENTITY LIST
        ******************
am-1.1> l in                          # list instances
        INSTANCES
        *********
        END OF INSTANCES
am-1.1> l ds                          # list definition store (D)

        DEFINITION STORE
        ****************
        END OF DEFINITION STORE
        ***********************
```

[19]Note: on line 2, cat is a standard UNIX command, not a feline.

```
am-1.1> l as                          # list action store (A)
        ACTION STORE
        ************
        END OF ACTION STORE
        ******************
am-1.1> cat ()                        # instantiate new cat entity
instantiating cat
am-1.1> l in
        INSTANCES
        *********
cat ()
        END OF INSTANCES
am-1.1> l ds

        DEFINITION STORE
        ****************
Variable # 1: height = 2
Variable # 2: pos = 2
Variable # 3: obstruct = 0
Variable # 4: intention = -1
Variable # 5: pushOut = intention>0&&obstruct
Variable # 6: pushIn = intention<0&&obstruct
        END OF DEFINITION STORE
        ***********************
am-1.1> l as

        ACTION STORE
        ************
Action # 1: intention>0&&!obstruct print("Cat moves outward") ->
        pos = |pos|+1
Action # 2: intention<0&&!obstruct print("Cat moves inward") ->
        pos = |pos|-1
        END OF ACTION STORE
        ******************
am-1.1> ?(pos)                        # show current definition and value
pos is defined as 2
pos evaluates to 2
am-1.1> set iterations = 4            # limit to 4 major transitions only
am-1.1> start                         # start the machine executing
starting simulation
#
 Cat moves inward
#
 Cat moves inward
#
 Cat moves inward
#
 Cat moves inward
* 4 iterations successfully completed
```

```
am-1.1> ?(pos)                          # cat is now inside the house
pos is defined as -1-1
pos evaluates to -2
                                        # cat now wants to go out
am-1.1> define intention = 1;
defining intention
am-1.1> flap()                          # instantiate the cat flap entity
instantiating flap
am-1.1> define obstruct = pos == 0 && angle < 3;
                                        # ideally this is defined in terms of
                                        # lflap * tan(angle), pos and height,
                                        # but am does not have tan()
defining obstruct
am-1.1> l ds                            # we now have definitions from flap

        DEFINITION STORE
        ****************
Variable # 1: height = 2
Variable # 2: pos = -1-1
Variable # 3: obstruct = pos==0&&angle<3
Variable # 4: intention = 1
Variable # 5: pushOut = intention>0&&obstruct
Variable # 6: pushIn = intention<0&&obstruct
Variable # 7: lflap = 5
Variable # 8: angle = 0
Variable # 9: switch = TRUE
Variable # 10: fourWayLock = 0
Variable # 11: Radius = 10
Variable # 12: elecLock = (pos>Radius||pos<-Radius)&&switch
Variable # 13: canPushOut = angle!=0||(!elecLock&&
                (fourWayLock==0||fourWayLock==1))
Variable # 14: canPushIn = angle!=0||(!elecLock&&
                (fourWayLock==0||fourWayLock==2))
        END OF DEFINITION STORE
        ***********************
am-1.1> set iterations = 8
am-1.1> start                           # output from major transitions
                                        # are separated by '#' characters.
                                        # Note the flap descends in parallel
                                        # with the cat motion
starting simulation
#
 Cat moves outward
#
 Cat moves outward
#
 Angle becomes 1
#
 Angle becomes 2
#
```

```
 Angle becomes 3
#
 Cat moves outward
 Angle becomes 2
#
 Cat moves outward
 Angle becomes 1
#
 Cat moves outward
 Angle becomes 0
* 8 iterations successfully completed
am-1.1> man()
instantiating man
am-1.1> start                          # man randomly acts on the flap
starting simulation
#
 Cat moves outward
#
 Cat moves outward
#
 Cat moves outward
 Four way lock is set to 1
#
 Cat moves outward
#
 Cat moves outward
#
 Cat moves outward
#
 Cat moves outward
 Switch off
#
 Cat moves outward
* 8 iterations successfully completed
am-1.1> ^D                             # exit am
$
```

Listing 2.6: An interaction with am and Y.P. Yung's cat flap