Lecture Th1: The Abstract Definitive Machine

Th1.1: Background and Motivation

The Abstract Definitive Machine (ADM) was designed by Beynon, Slade and Yung in 1988. It was developed in collaboration with Mark Norris at British Telecom as part of an investigation into animation tools based on LSD. The first implementation was due to Slade. The ideas described here are based on Slade's MSc thesis [] and related papers [].

The ADM is significant in our research as a step towards a general-purpose definitive programming model. The practical programming techniques we use in supporting definitive scripts don't give us much insight into what such a model should be, whilst agent-oriented LSD specification is – in itself – not to be regarded as an operational model. In this section, we briefly explore some of the issues, then describe the current status of the ADM design and implementation. We shall also consider the appropriate direction for future development of the ADM concept.

The Practical Programming Perspective:

The computational model of eden is in some ways unsatisfactory. It includes features such as actions that describe state by side-effect that can be easily abused. There is no built-in means to discipline actions, so that the sequence in which actions are executed can become disorderly. SCOUT demonstrates that definitive specification of the screen is the most appropriate complement for definitive representations of internal state. Since hardware doesn't operate in a definitive way – at any rate at the appropriate level of abstraction – we need to use implementation techniques such as eden provides, but we'd like to conceive them in a more elegant and formally satisfactory framework. The ADM was designed as a computational model that is based upon definitive representations of state. One motivation behind its design is thus:

Is there a good abstract model for "proper use" of eden?

[Clearly there are many possible ways to abuse eden, whereby procedures and rules displace any principled use of definitions to describe state.]

The Animation Perspective:

An independent motivation for the ADM comes from concurrent systems modelling. The ADM was primarily developed as a tool for animation of LSD specifications. There are two principal themes behind animation with definitive representations of state:

1) the representation of concurrent action by parallel redefinition
2) the use of scripts to reflect context dependence of agent actions.

The design and simulation activities illustrated in the VCCS show the potential for rich forms of interaction, where the system designer can observe an intelligible state-based representation of a concurrent system, and is empowered to intervene and redirect computation to an unusual degree. The ADM is a first step towards describing an abstract computational model with these characteristics.

Th1.2: The ADM state-transition model

The ADM model in some ways resembles a traditional procedural model of computation. To develop the analogy, a Pascal program is essentially defined by procedures, instances of which are placed on the stack during execution. The values that are of computational interest to the programmer are represented on the stack, but there is also a stored program that determines the sequence by which the values of instantiated variables are modified and procedural instances are created and destroyed. We shall use the term "computational state" to refer to the values of variables that are on the execution stack during the computation: these are the variables whose values the programmer inspects when debugging.

There is a loose analogy between the ADM and a Pascal interpreter:

	computational state	*program*	*primitive operation*
Pascal	variables on stack	set of procedures	assign variable call procedure
ADM	variables in script	set of entities	redefine variable create/delete entity

To explain more fully: an ADM program is a set of entities (cf procedures) and on execution a set of entities is instantiated. Each entity comprises a set of definitions and a set of actions. Each action is a guarded sequence of primitive operations, viz: the redefinition of a variable or the creation or deletion of an entity instance. During execution of the ADM program, the definitions associated with currently instantiated entities are stored in the definition store D (cf the execution stack). The actions associated with an entity are stored in the action store A (cf the stored program). In each machine cycle, the computational state as defined by the definiton store D is modified according to the contents of the action store A (cf the way in which the contents of the execution stack are changed according to the contents of the current instruction in the program store). Figure 1 illustrates the relationship between the program entities, the definition store and the action store during the execution of an ADM program.

<Figure to be introduced at this point>

For the present, with the above analogy in mind, it is convenient to discuss the ADM in terminology that suggests a conventional abstract machine model (as in []) – beware though that the next lecture will qualify this interpretation and indicate where it can be fundamentally misleading.

It is also helpful at this point to view the ADM design from the perspectives mentioned in Th1.1:

- where the abstract description of sound eden programming is concerned, it is clear that entities are well-suited to the abstract representation of definitive notation implementation in eden. For instance, the definition of a DoNaLD point leads to the creation of a definition in eden together with an associated procedural eden action that is responsible for maintaining the visualisation of the point. This indicates how geometric entities in DoNaLD might correspond to ADM entities in modelling the eden implementation of DoNaLD.

- where the animation of an LSD specification is concerned, it is clear that the concept of entity is in form quite similar to an agent specification. The fundamental difference is that the privileges of an LSD agent do not bear a direct computational interpretation; they represent potential actions rather than obligations to act.

As a final point of comparison, note that the ADM execution differs from the Pascal model in that the content of the program store is modified dynamically as entities are created and deleted, whereas in Pascal the content of the program store is independent of the execution and the execution state is determined simply by the position of the program counter. The definitions and actions in the ADM also differ from their counterparts in EDEN, which are persistent, rather than ephemeral.

Th1.3: Characteristics of the ADM

The ADM machine cycle:

Each ADM machine cycle follows the pattern:

consult values in context of D the guards of actions in A	perform in parallel all actions in A with true guards

Note that each action is a sequence of primitive operations. It is assumed that the way in which primitive operations synchronise is not significant. Instances of parallel redefinition of the same variable, parallel definition that leads to cyclicity, or redefinitions that involve the evaluation of variables that are themselves to be redefined in a parallel action (etc) are detected as interference between actions. The default operation of the machine in the presence of interference is to suspend the computation.

Programming the ADM:

What are appropriate ways to program the ADM?

The ADM represents an abstract state-based programming paradigm with parallelism. An example of a simple program that can be expressed using the ADM is the implementation of a systolic array (cf []). Such an implementation describes the states of the computation in a most appropriate way, expressing the synchronisation between operations in the array components, but not eliminating state as in a pure data flow model.

As yet, the abstract programming potential of the ADM has been little explored. There are several possible reasons for this, including the limitations of the current implementation (see below) and design. The blocks.am example illustrates some of the relevant issues.

The primary use of the ADM has been in connection with animation from LSD specifications. Though the ADM was originally developed to complement LSD, it is also possible to regard the use of LSD as a software development method for the ADM.

The ADM in use:

Execution in the ADM has many of the abstract characteristics we have already illustrated e.g. in the VCCS. The ADM programmer can be regarded as an agent with exceptional privileges that include the power to introduce an arbitrary action in any machine cycle. (In this role, the programmer resembles the debugger of a Pascal program in an environment that allows variables to be freely inspected and reassigned.) This is one way in which problems of interference can be resolved – through special action on the part of the programmer to arbitrate where there is ambiguity about which actions should be selected for execution.

The scope for automatic detection of interference and singular conditions in execution is a powerful feature that can be exploited in several ways. In principle, it creates an environment for simulation in which the designer has unusual scope to intervene and redirect during execution. Call-by-need input is one of the mechanisms that can be naturally expressed in the ADM framework: when an undefined variable is encountered, the designer can be prompted to enter a suitable definition of its value.

Current Status and Limitations of ADM Design and Implementation:

The basic ADM, as implemented by Slade [], has only integer data types. This has been very limiting where animation is concerned. The addition of a textual output mechanism provides a partial solution to this problem. An action g -> A can be given a textual annotation T(...) using the contruct:

g {T(...)} -> A

– this has the effect of printing the string T(...) whenever the action is executed. [Here (...) is an informal notation for a parameter list, consisting of references to current variables in D.]

By using the textual output mechanism we can construct a commentary on a simulation – as illustrated by the Railway Station Animation. This is not a particularly satisfactory form of output, as it represents state through the accumulation of procedural actions rather than in a definitive style.

In principle, the ADM model should admit definitions rich enough to allow the visualisation to be expressed using definitive notations such as SCOUT and DoNaLD. The textual output mechanism can emulate this in a clumsy way by generating definitions in these notations that can be piped to the SDE system (cf blocks.am). Recent work of Simon Yung has addressed this issue, which is central to devising a good abstract representation for the principled use of EDEN.

The design of the ADM, as described above, raises a more fundamental concern. The sharp-eyed reader may observe that the mechanisms by which entities are created and destroyed are procedural rather than definitive in nature: this is a limitation with important practical implications. For instance, we may wish to link the presence of entities to some condition of the system in an indivisible fashion <example>. The definitions of the entities that make up the ADM program might also be more appropriately specified using definitive methods. For instance, we might wish to specify that the geometric symbol representing a particular geographic feature on a map changes according to the resolution. There are often ways to fudge these issues using existing tools, but there a good reasons to suppose that better abstractions have to be sought in these areas.