Lecture Th1: The Abstract Definitive Machine

Th1.1: Background and Motivation

The Abstract Definitive Machine (ADM)

due Beynon, Slade and Edward Yung in 1988.

Motivated by

- collaborative work with BTRL: Beynon, Norris (BTRL), Slade
- issues in Eden programming: Beynon, Edward Yung

ADM aimed at a general-purpose definitive programming model

Eden doesn't give us much insight into what the model should be

agent-oriented LSD specification not an operational model

Consider

- current status of the ADM design and implementation
- future development of the ADM concept.

The Practical Programming Perspective:

Computational model of eden abstractly unsatisfactory.

- actions describe state by side-effect: easily abused
- no built-in means to discipline actions: disorderly sequencing

Many possible ways to abuse eden

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

Definitive specification of the screen best way to
complement definitive representations of internal state

Practical need for procedural techniques as in EDEN

BUT need more elegant and formally satisfactory framework

ADM = computational model with definitive representation of state

The Animation Perspective:

Independent motivation for ADM: concurrent systems modelling

ADM = a tool for animation of LSD specifications

Two themes in animation with definitive representations of state:

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

ADM = abstract computational model allowing rich interaction

cf. design and simulation activities illustrated in the VCCS

Th1.2: The ADM state-transition model

ADM model <--> a traditional procedural model of computation

Execution of Pascal program involves

- procedure instances on the execution heap
  - – includes values of computational interest to the programmer

- a stored program to describe

  how values of instantiated variables are modified

  how procedural instances are created and destroyed

Use "computational state" to refer to

values of variables on the execution heap during execution

values of these variables are inspected in debugging.

Loose analogy <--> ADM and a Pascal machine:

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

An ADM program is a set of entities (cf procedures)

– on execution a set of entities is instantiated

Each entity comprises a set of definitions and a set of actions

Each action = guarded sequence of primitive operations, viz: variable redefinition or creation/deletion of an entity instance

In ADM execution

- definitions in currently instantiated entities

  are stored in the definition store D (cf the execution heap)

- actions in currently instantiated entities

  are stored in the action store A (cf the stored program)

In each machine cycle:

computational state as defined by D is modified

according to contents of the action store A

cf contents of the execution heap are changed

according to current instruction in the program store

NB from above analogy, can view ADM

as a conventional abstract machine model

Beware: *later* will need to

qualify this interpretation – can be fundamentally misleading.

ADM design in perspective

.... sound eden programming?

- entities suit abstract representation

  of definitive notation implementation in eden

  E.g. definition of a DoNaLD point leads to creation of

  - a definition in eden
  - an associated procedural eden action to maintain point

  Hence geometric entities in DoNaLD <--> ADM entities

  in modelling the eden implementation of DoNaLD.

- for animation of an LSD specification

  form of entity similar to an LSD agent specification

  BUT fundamental distinction:

  LSD agent privileges = potential actions not obligations to act.

ADM execution unlike Pascal / EDEN models:

- content of the program store is modified dynamically

  cf content of the program store independent of the execution execution state <--> position of program counter

- definitions and actions in the ADM do not persist

  cf counterparts in EDEN, harder to manipulate in execution

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

Each action is a sequence of primitive operations

Assume synchronisation of primitive operations not significant

Forms of interference detected include:

- parallel redefinition of the same variable
- parallel definition that leads to cyclicity
- redefinitions that involve the evaluation of variables that are themselves redefined in a parallel action

Default for ADM on interference is to suspend the computation.

Programming the ADM:

What are appropriate ways to program the ADM?

ADM = abstract state-based parallel programming paradigm

Simple sample program is implementation of a systolic array:

describes computational state in an appropriate way

expressing synchronisation between the array components

BUT not eliminating state as in a pure data flow model.

ADM little explored as abstract programming model:

issues include limitations of current implementation & design

cf blocks.am example

Two views

ADM for animation from LSD specifications

LSD as a software development method for the ADM.

The ADM in use:

Powerful features in execution (cf the VCCS demo)

- ADM programmer can act as agent with exceptional privileges

  e.g. can introduce an arbitrary action in any machine cycle.

  cf debugging Pascal where vars freely inspected & reassigned

  => programmer can arbitrate to resolve interference

- automatic detection of interference and singular conditions

  => programmer can intervene and redirect during execution

- Call-by-need input: if an undefined variable is encountered, prompt programmer for suitable definition of its value

Current Status and Limitations of the ADM

Implementation

Basic ADM, as implemented by Slade, only integer data types

Communication with other system via textual output mechanism:

– an action g -> A can generate text T(...) via the construct:

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

that prints the string T(...) whenever the action is executed

Textual output mechanism generates commentary:

– this represents state by

accumulation of procedural actions

rather than in a definitive style.

In principle, ADM definitions should be in SCOUT and DoNaLD.

- can emulate by piping defns to the SDE system (cf blocks.am)
- can use ADM to EDEN translation, as devised by Simon Yung

Current Status and Limitations of the ADM

Design

- creating/deleting entities procedural not definitive in style

  => can't link presence of entities to some condition indivisibly

  cf LIVE derivates in LSD specifications

- definitions of the entities aren't specified definitively

  => can't dynamically redefine entity cf labels in VCCS speedo

References

W M Beynon, M D Slade, Y W Yung
Parallel Computation in Definitive Models CONPAR 88
W M Beynon
Definitive Programming for Parallelism Parallel Computing 89
M D Slade
Parallel Definitive Programming MSc thesis, Warwick Univ 1989