The Abstract Definitive Machine

(See Lectures 5 and 7 in the EM for Concurrency series)

Many perspectives ….

- definitive parallel programming
- animation of LSD accounts
- conceptual framework for EM in EDEN

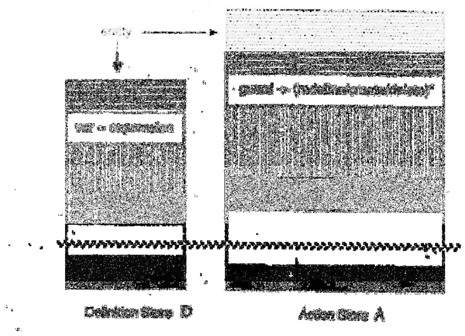
- machine-computing-oriented viewpoint
- human-computing-oriented viewpoint

Many variants ….

- first design / implementation Slade 1990
- various translators from ADM to EDEN
  - Y P Yung, P-H Sun
- underlying concept obscured, recovered by Ward: *The Authentic ADM* (2004)

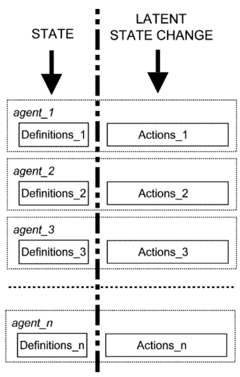
Basic architecture of the ADM

The Abstract Definitive Machine: entity = definitions + actions



Linking LSD agents to ADM entities …

LSD agents' state and derivate observables



LSD agents' protocols

Core features of the ADM 1

- entity = set of definitions + set of actions
- have instances of abstract entities
- action is guarded sequence of form: (redefinition + entity invocation/deletion)*
- in some contexts actions have been interpreted as *atomic*; better conceived as *interleaving asynchronously*

Core features of the ADM 2

- model of “true” concurrent interaction
- definitions can be performed in parallel
- scope for syntactic checks on interference
- changes of state admit free interpretation:
  - “computational step” in machine
  - redesign / reprogramming step
  - manual, automated and semi-automated

Some illustrative examples

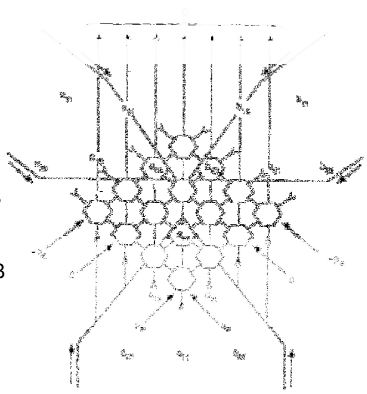
A systolic array simulation

Parallel programming

Model of systolic array for sparse matrix multiplication …

Input matrices A and B at top-left and top-right

Output matrix C = A . B at top centre



EM paper 010: *Parallel computation in definitive models*

The Railway Station Animation

LSD account of the stationmaster

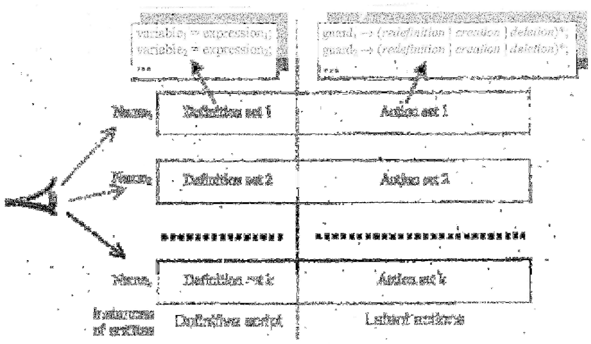
```
agent sm() {

oracle  (time) Limit, Time,            // knowledge of time to elapse before departure due
        (bool) guard_raised_flag,      // knowledge of whether the guard has raised his flag
        (bool) driver_ready,           // knowledge of whether the driver is ready
        (bool) around[d],              // knowledge of whether there's anybody around doorway
        (bool) door_open[d];    // the open/close status of door d (for d = 1 .. number_of_doors)

state   (time) tarrive = |time|,       // the S-M registers time of arrival
        (bool) can_move,               // the signal observed by driver for starting engine
        (bool) whistle = false,        // the whistle is not blowing
        (bool) whistled = false,       // the whistle has not blown
        (bool) sm_flag = false,        // S-M lowers flag
        (bool) sm_raised_flag = false; // S-M has not raised flag

handle  (bool) can_move,
        (bool) whistle,
        (bool) whistled,
        (bool) sm_flag,
        (bool) sm_raised_flag;
        (bool) door_open[d];    // the open/close status of door d (for d = 1 .. number_of_doors)

derivate
                        number_of_doors
        (bool) ready =      /\   (!door_open[d]);        // are all doors shut?
                          d = 1
        (bool) timeout = (Time - tarrive) > Limit;      // departure due

protocol
        door_open[d] ^ !around[d] -> door_open[d] = false; (d = 1 .. number_of_doors)
        ready ^ timeout ^ !whistled -> whistle = true; whistled = true; guard(); whistle = false;
        ready ^ whistled ^ !sm_raised_flag -> sm_flag = true; sm_raised_flag = true;
        sm_flag ^ guard_raised_flag -> sm_flag = false;
        ready ^ guard_raised_flag ^ driver_ready ^ engaged ^ !can_move -> can_move = true;

}
```

```
entity sm() {
definition
    whistle = false, whistled = false, sm_flag = false,
    sm_raised_flag = false, can_move = false,
    ready is  !door_open{1} && !door_open{2}, tarrive,
    Limit = 20, timeout is (Time - tarrive) > Limit,
    level = 0, init = true
action
    init → tarrive = Time; init = false,
    door_open{1} && !around{1}
        print("Station master shuts door 1")
         → door_open{1} = false,
    ……
}
```

```
entity sm() {
definition
    whistle = false,
    whistled = false,
    sm_flag = false,
    sm_raised_flag = false,
    can_move = false,
    ready is  !door_open{1} && !door_open{2},
    tarrive,
    Limit = 20,
    timeout is (Time - tarrive) > Limit,
    level = 0,
    init = true
action
        ……
}
```

```
entity sm() {
definition
    ….
action
    init → tarrive = Time; init = false,
    door_open{1} && !around{1}            print("Station master shuts door 1")
          → door_open{1} = false,
    door_open{2} && !around{2}            print("Station master shuts door 2")
          → door_open{2} = false,
    ready && timeout && !whistled     print("Station master whistles to call guard")
          → whistle = true; whistled = true; guard(); level = 1,
    level == 1 print("Station master stops whistling") → whistle = false; level = 0,
    ready && whistled && !sm_raised_flag  print("Station master raises his flag")
          → sm_flag = true; sm_raised_flag = true,
    sm_flag && guard_raised_flag          print("Station master lowers his flag")
          → sm_flag = false,
    ready && guard_raised_flag && driver_ready && engaged && !can_move
       print("Train can move now") → can_move = true
}
```

Human and Machine Perspectives on the ADM



The ADM from a machine perspective (Rungrattanaubol, 2002)

Machine perspective on ADM

Machine-like execution:
- true guard as obligation to perform action
- action performed automatically / atomically

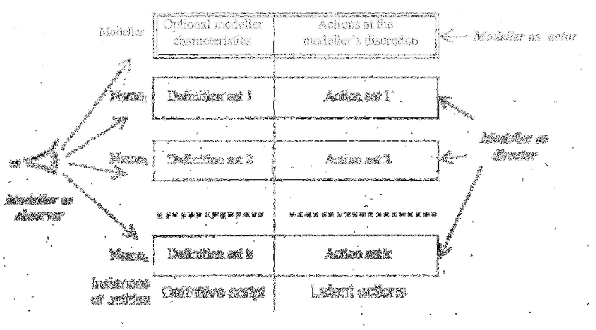
Examples
- systolic array
- railway station animation
- telephone animation

About the examples

Systolic array ✓
- highly structured, synchronised, clocked

Railway station animation ×
- too regimented, clock cycle metaphor
  "init → tarrive = |Time|; init = false" is atomic

Telephone animation ?
- embellish actions with probabilities to reflect delay, timeliness of response; introduces artificial observables / actions



The ADM from the human perspective (Rungrattanaubol, 2002)

Human perspective on ADM

"Free agent" style execution (cf. AADM):
- true guard as entitlement to perform action
- action not atomic – intermediate states
- re-evaluation of guards during execution

Further from implementation …
execution / interpretation needs human input
… modeller takes a role in directing / acting
necessary to capture semantics of EDEN use

JUGS in the ADM

	Entity name	Definition set	Action set
Human agents	Pupil		*Button1 valid → select button1*; …
	Teacher		*true* → capA, capB, target=…; …
Interface elements	Button1	button1_colour is valid1? "white": "black"; button1_label is "1: FillA"; button1_position is …	valid1 → input =1;
	Button2	Button2_colour is valid2? "white": "black"; …	valid2 → input =2;
	JugAwin	*Scout definition of a window to display JugA*	
	JugBwin	*Scout definition of a window to display JugB*	
	< Supplementary alternative visualisation e.g. for text message on mobile phone >		
	JugAtxt	jugA is jugdisplay(height,capA,widthA,contentA);	*jugA touched → redisplay jugA;*
Internal state-transition model	init_pour	input is 1; $LIVE_{pour}$ is updating;	*input touched* → step , option , updating = 1, …, 1;
	pour {option}	input is 1;	valid1 → contentA=contentA+1;
External entity-relationship model	JugA	capA=5; contentA=3; Afull is capA==contentA;	
	JugB	capB=2; contentB=2; Bfull is capB==contentB;	
	Environ-ment	valid1 is !Afull; valid4 is contentB != 0; valid7 is valid1 && valid4;	

The *Authentic* Abstract Definitive Machine

Ashley Ward
(after Beynon and Slade)

Ambiguity in ADM writings

The description of the ADM in Slade refers to execution in which sequences of commands in ADM actions are executed atomically

This originates from the need to cope with instantiating observables and initialising entities, or resetting observables and deleting entities, *in a single step*

In the application of the ADM in 'animating LSD', it is appropriate to think of a guard as a cue that enables an entity to initiate a sequence of actions to be performed asynchronously: this is the execution model for what Ward terms the *authentic* ADM …

Execution model for the Authentic ADM

In each step:
(The state is now S)
For each action *a*:
    If the action *a* is currently executing and there is no command from *a* already in the runset (pending execution)
        Add the next command in action *a* to the runset
    Else:
        Evaluate guard of *a* in state S
        If guard of *a* is true:
            Add the first command in action *a* to the runset
Check the runset for an invalid transition
If the transition is invalid,
    Stop and ask the modeller to resolve the conflict before proceeding
Select a subset of the commands from the runset and execute these, conceptually in parallel, making a transition to the state S'
(The state is now S')

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 runset (pending execution)
        4. Add the next command in action *a* to the runset
    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 runset
9. Check the runset 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 runset and execute these, conceptually in parallel, making a transition to the state S'
13. (The state is now S')

Notes on selection of actions

At step 12, selection of the subset of commands can be determined non-deterministically by the algorithm or determined by the modeller in the 'super-agent' role

Due to the guarantee given by the invalid transition check, there is no interference between actions in the runset

… in an implementation

Commands can therefore be performed sequentially or in parallel

If commands are performed sequentially, the state will transit intermediate states before it reaches S'. Evaluations can be performed in these intermediate states or in S without influencing the result as there is no interference between commands.