DEDUCTIVE REASONING AGENTS

3.1 Agents as Theorem Provers
3.2 Agent-Oriented Programming
3.3 Concurrent MetateM

Agent Architectures

- An *agent* is a computer system capable of *flexible autonomous action…*
- Issues one needs to **address** in order to build agent-based systems…
- Three types of agent *architecture*:
  - symbolic/logical
  - reactive
  - hybrid

Agent Architectures

- We want to build agents, that enjoy the properties of **autonomy**, **reactiveness**, **pro-activeness**, and **social** ability that we talked about earlier
- This is the area of *agent architectures*
- **Agent architecture can be defined as:**

  '[A] particular methodology for building [agents].

  It specifies how… the agent can be decomposed into the construction of a set of component modules.

  How these modules should be made to interact.

  The total set of modules and their interactions has to provide an answer to the question of how the sensor data and the current internal state of the agent determine the actions and future internal state of the agent.

  An architecture encompasses techniques and algorithms that support this methodology.'

Agent Architectures

- Another definition of Agent architecture to be:

  ‘[A] specific collection of software (or hardware) modules, typically designated by boxes with arrows indicating the data and control flow among the modules.

  A **<u>more abstract view</u>** of an architecture is as a general methodology for designing particular modular decompositions for particular tasks.’

Agent Architectures

- Originally (1956-1985), much all agents designed within AI were ***symbolic reasoning*** agents
- Its purest expression proposes that agents use *explicit logical reasoning* in order to decide what to do
- Problems with symbolic reasoning led to a reaction against this — the so-called *reactive agents* movement, 1985–present
- From 1990-present, a number of alternatives proposed: *hybrid* architectures, which attempt to combine the **best of reasoning** and reactive architectures

Symbolic Reasoning Agents

- The classical approach to building agents is to view them as a particular type of knowledge-based system, and bring all the associated (discredited?!) methodologies of such systems to bear
- This paradigm is known as ***symbolic AI***
- We define a deliberative agent or agent architecture to be one that:
  - contains an explicitly represented, symbolic model of the world
  - makes decisions (for example about what actions to perform) via symbolic reasoning

Symbolic Reasoning Agents

- If we aim to build an agent in this way, there are two key problems to be solved:

1. *The transduction problem*:
   that of translating the **real world** into an accurate, adequate symbolic description, in time for that description to be useful…vision, speech understanding, learning
2. *The representation/reasoning problem*:
   that of **how to symbolically represent information about complex real-world** entities and processes, and how to get agents to reason with this information in time for the results to be useful…knowledge representation, automated reasoning, automatic planning

Symbolic Reasoning Agents

- Most researchers accept that neither problem is anywhere near solved
- Underlying problem lies with the complexity of symbol manipulation algorithms in general: many (most) search-based symbol manipulation algorithms of interest are *highly intractable*
- Because of these problems, some researchers have looked to alternative techniques for building agents; we look at these later

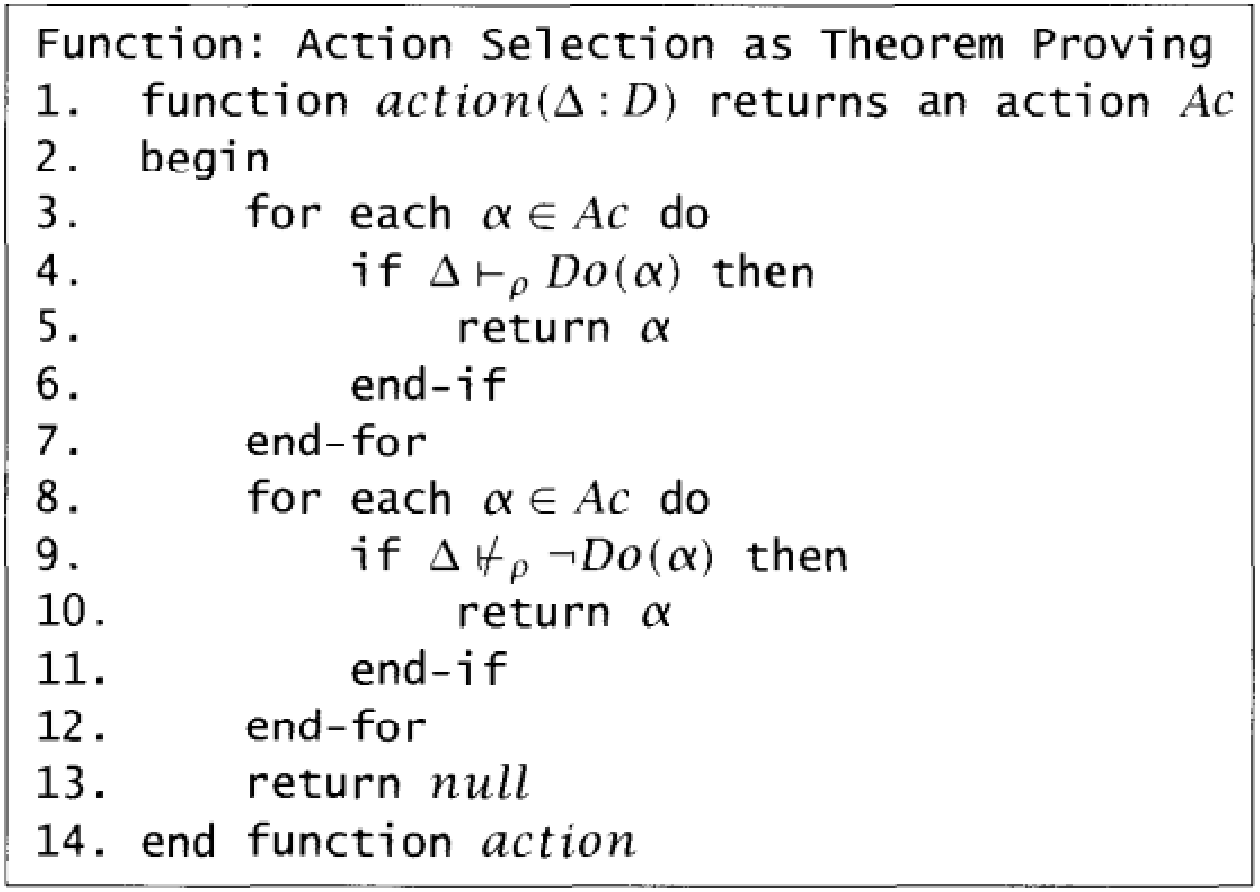
Deductive Reasoning Agents

- How can an agent decide what to do using theorem proving?
- Basic idea is to use **logic** to encode a theory stating the *best* action to perform in any given situation
- Let:
  - ρ be this theory (typically a set of rules)
  - Δ be a logical database that describes the current state of the world
  - Ac be the set of actions the agent can perform
  - Δ ✿$_{\rho}\phi$ mean that ϕ can be proved from Δ using ρ

Deductive Reasoning Agents

```
/* try to find an action explicitly prescribed */
for each $a \in Ac$ do
          if $\Delta \vdash_{\rho} Do(a)$ then
                    return $a$
          end-if
end-for
/* try to find an action not excluded */
for each $a \in Ac$ do
          if $\Delta \nvdash_{\rho} \neg Do(a)$ then
                    return $a$
          end-if
end-for
return $null$ /* no action found */
```

Deductive Reasoning Agents

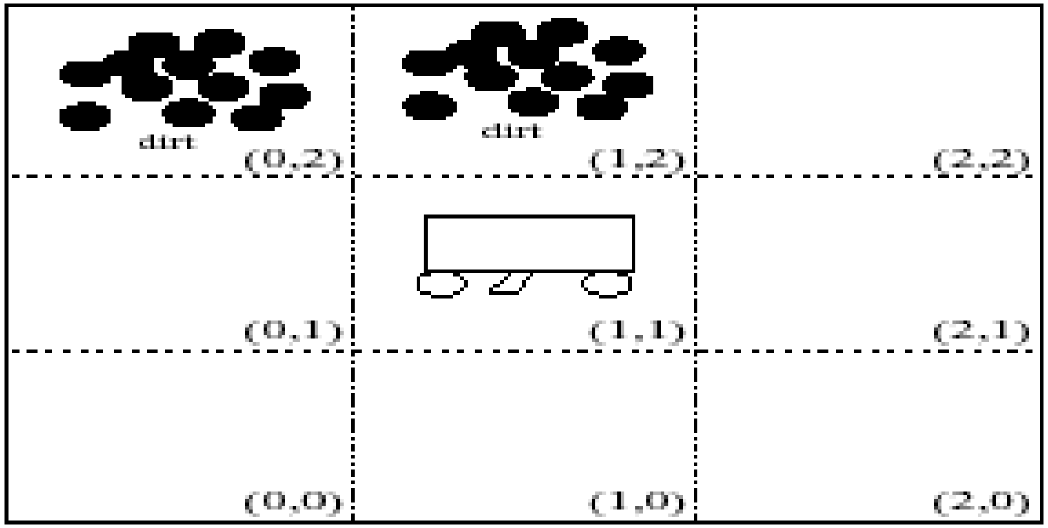


```
Function: Action Selection as Theorem Proving
1.  function $action(\Delta : D)$ returns an action $Ac$
2.  begin
3.      for each $\alpha \in Ac$ do
4.          if $\Delta \vdash_\rho Do(\alpha)$ then
5.              return $\alpha$
6.          end-if
7.      end-for
8.      for each $\alpha \in Ac$ do
9.          if $\Delta \nvdash_\rho \neg Do(\alpha)$ then
10.             return $\alpha$
11.         end-if
12.     end-for
13.     return $null$
14. end function $action$
```

Figure 3.2 Action selection as theorem-proving.

Deductive Reasoning Agents

- An example: The Vacuum World
- Goal is for the robot to clear up all dirt



Deductive Reasoning Agents

- Use 3 *domain predicates* to solve problem:

$In(x, y)$	agent is at (x, y)
$Dirt(x, y)$	there is dirt at (x, y)
$Facing(d)$	the agent is facing direction d

- Possible actions:

$$Ac = \{turn, forward, suck\}$$

P.S. *turn* means “turn right”

Deductive Reasoning Agents

- Rules ρ for determining what to do:

$$
\begin{aligned}
In(0,0) \wedge Facing(north) \wedge \neg Dirt(0,0) &\longrightarrow Do(forward) \\
In(0,1) \wedge Facing(north) \wedge \neg Dirt(0,1) &\longrightarrow Do(forward) \\
In(0,2) \wedge Facing(north) \wedge \neg Dirt(0,2) &\longrightarrow Do(turn) \\
In(0,2) \wedge Facing(east) &\longrightarrow Do(forward)
\end{aligned}
$$

- …and so on!
- Using these rules (+ other obvious ones), starting at (0, 0) the robot will clear up dirt

Deductive Reasoning Agents

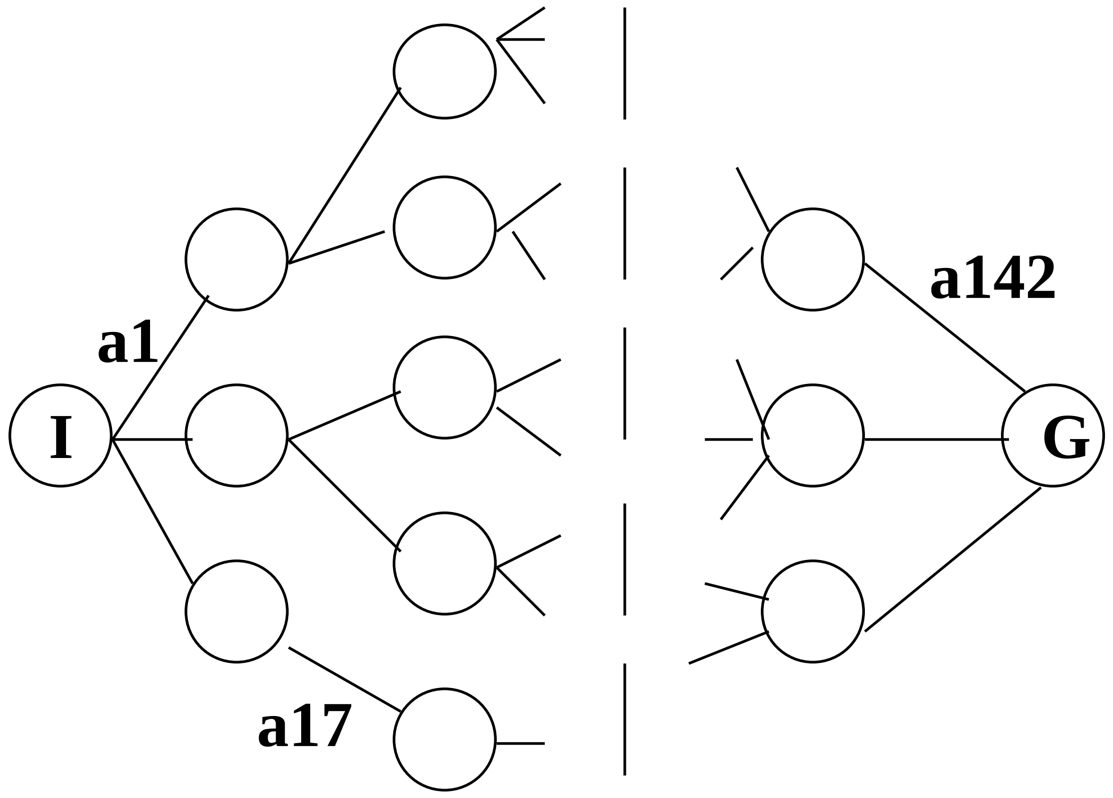
- Problems:
  - How to convert video camera input to $Dirt(0, 1)$?
  - decision making assumes a *static* environment: *calculative* rationality
  - decision making using first-order logic is *undecidable*!
- Even where we use *propositional* logic, decision making in the worst case means solving co-NP-complete problems
  (PS: co-NP-complete = bad news!)
- Typical solutions:
  - weaken the logic
  - use symbolic, non-logical representations
  - shift the emphasis of reasoning from *run time* to *design time*

More Problems…

- The “logical approach” that was presented implies adding and removing things from a database
- That’s not pure logic
- Early attempts at creating a “planning agent” tried to use true logical deduction to the solve the problem

Planning Systems (in general)

- Planning systems find a sequence of actions that transforms an initial state into a goal state



Planning

- Planning involves issues of both Search and Knowledge Rrepresentation
- Sample planning systems:
  - Robot Planning (STRIPS)
  - Planning of biological experiments (MOLGEN)
  - Planning of speech acts
- For purposes of exposition, we use a simple domain – The Blocks World

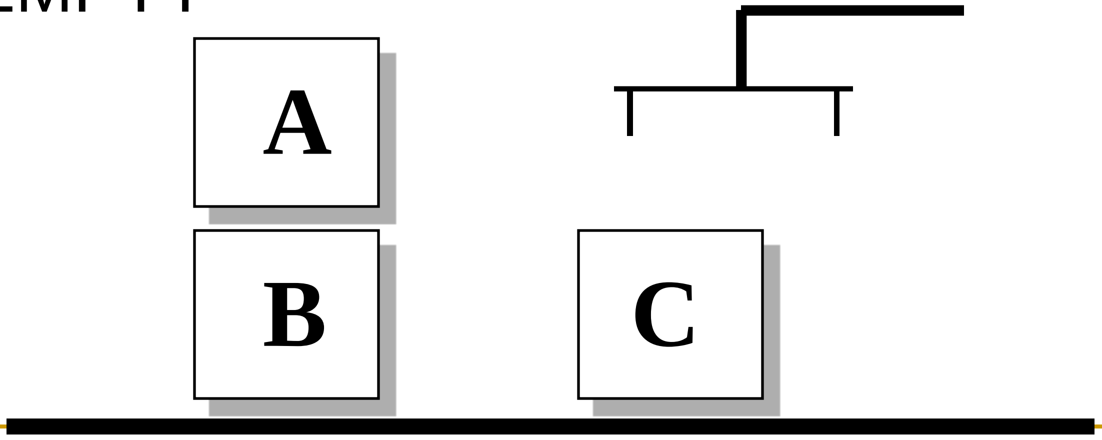
The Blocks World

- The Blocks World (today) consists of equal sized blocks on a table
- A robot arm can manipulate the blocks using the actions:
  - UNSTACK(a, b)
  - STACK(a, b)
  - PICKUP(a)
  - PUTDOWN(a)

The Blocks World

- We also use predicates to describe the world:
  - ON(A,B)
  - ONTABLE(B)
  - ONTABLE(C)
  - CLEAR(A)
  - CLEAR(C)
  - ARMEMPTY

In general:
ON(a,b)
HOLDING(a)
ONTABLE(a)
ARMEMPTY
CLEAR(a)



Agent Oriented Programming

AGENT0 and PLACA

- Much of the interest in agents from the AI community has arisen from Shoham's notion of *agent oriented programming* (AOP)
- AOP a 'new programming paradigm, based on a societal view of computation'
- The key idea that informs AOP is that of directly programming agents in terms of intentional notions like belief, commitment, and intention
- The motivation behind such a proposal is that, as we humans use the intentional stance as an *abstraction* mechanism for representing the properties of complex systems.
  In the same way that we use the intentional stance to describe humans, it might be useful to use the intentional stance to program machines.

AGENT0

- Shoham suggested that a **complete AOP system will have 3 components:**
  - a logic for specifying agents and describing their mental states
  - an interpreted programming language for programming agents
  - an ‘agentification’ process, for converting ‘neutral applications’ (e.g., databases) into agents
- Results only reported on first two components.
- Relationship between logic and programming language is *semantics*
- We will skip over the logic(!), and consider the first AOP language, AGENT0

AGENT0

- AGENT0 is implemented as an extension to LISP
- Each agent in AGENT0 has 4 components:
  - a set of capabilities (things the agent can do)
  - a set of initial beliefs
  - a set of initial commitments (things the agent will do)
  - a set of *commitment rules*
- The key component, which determines how the agent acts, is the commitment rule set

AGENT0

- Each commitment rule contains
  - a *message condition*
  - a *mental condition*
  - an action
- On each ‘agent cycle’…
  - The message condition is matched against the messages the agent has received
  - The mental condition is matched against the beliefs of the agent
  - If the rule fires, then the agent becomes committed to the action (the action gets added to the agent’s commitment set)

AGENT0

- Actions may be
  - *private*:
    an internally executed computation, or
  - *communicative*:
    sending messages
- Messages are constrained to be one of three types:
  - “requests” to commit to action
  - “unrequests” to refrain from actions
  - “informs” which pass on information

AGENT0

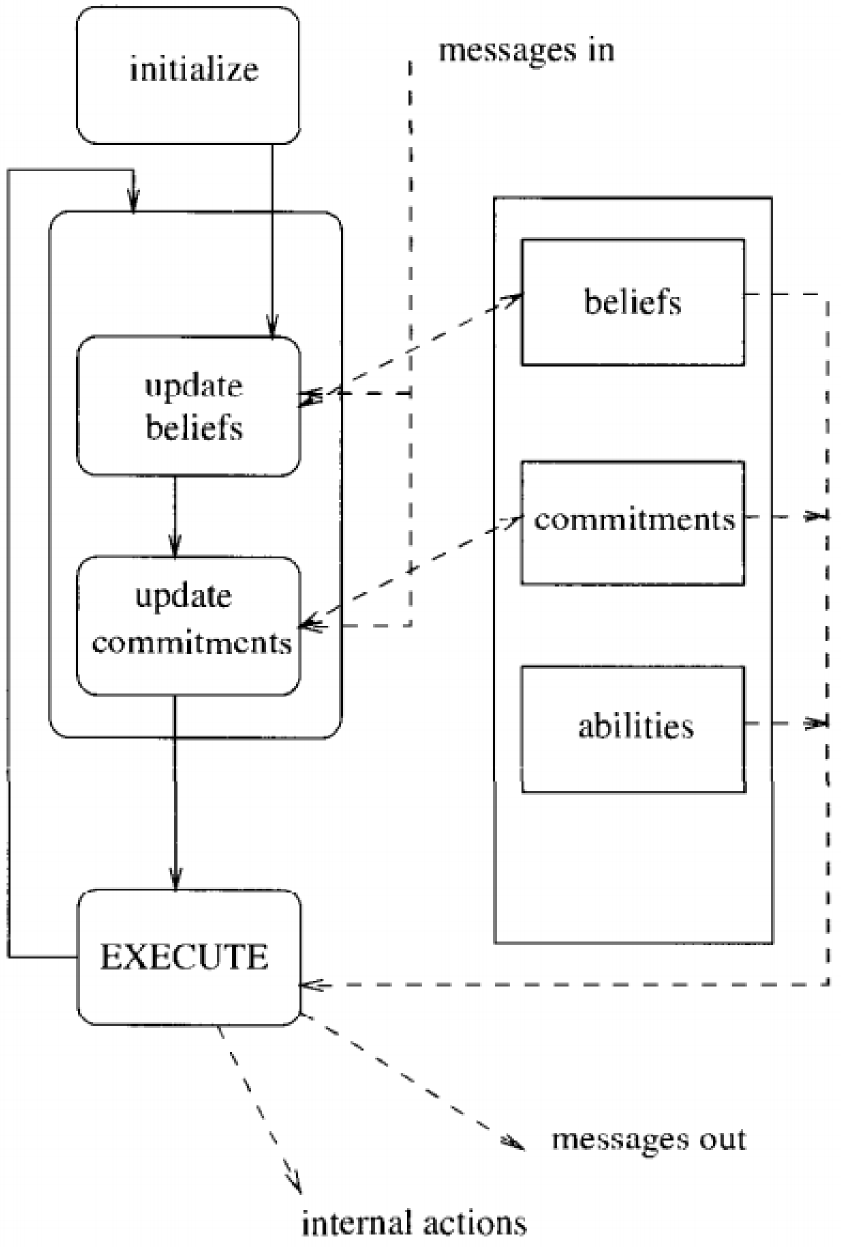


Figure 3.4 The flow of control in Agent0.

AGENT0

- A commitment rule:

```
COMMIT(
      ( agent, REQUEST, DO(time, action)
      ), ;;; msg condition
      ( B,
            [now, Friend agent] AND
            CAN(self, action) AND
            NOT [time, CMT(self, anyaction)]
      ), ;;; mental condition
      self,
      DO(time, action)
)
```

AGENT0

- This rule may be paraphrased as follows:
  if I receive a message from *agent* which requests me to do *action* at *time*, and I believe that:
  - *agent* is currently a friend
  - I can do the action
  - At *time*, I am not committed to doing any other action

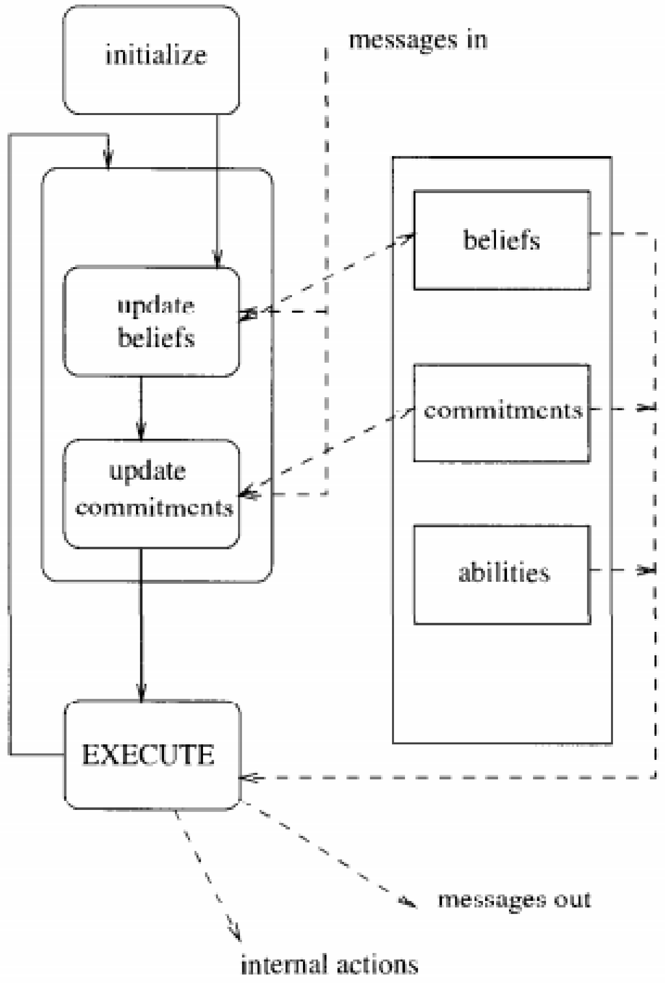
  then commit to doing *action* at *time*

The operation of an agent can be described by the following loop (see Figure 3.4).

(1) Read all current messages, updating beliefs – and hence commitments – where necessary.

(2) Execute all commitments for the current cycle where the capability condition of the associated action is satisfied.

(3) Goto (1).



AGENT0 and PLACA

- AGENT0 provides support for multiple agents to cooperate and communicate, and provides basic provision for debugging…
- …it is, however, a *prototype*, that was designed to illustrate some principles, rather than be a production language
- A more refined implementation was developed by Thomas, for her 1993 doctoral thesis
- Her Planning Communicating Agents (**PLACA**) language was intended to address one severe drawback to AGENT0: the **inability of agents to plan, and communicate requests** for action via high-level goals
- Agents in PLACA are programmed in much the same way as in AGENT0, in terms of *mental change* rules

Concurrent METATEM

Concurrent METATEM

- Concurrent METATEM is a multi-agent language in which each agent is programmed by giving it a ***temporal logic*** specification of the behavior it should exhibit
- These specifications are executed directly in order to generate the behavior of the agent
- Temporal logic is classical logic augmented by *modal operators* for describing how the truth of propositions **changes over time**

Concurrent METATEM

- For example. . .

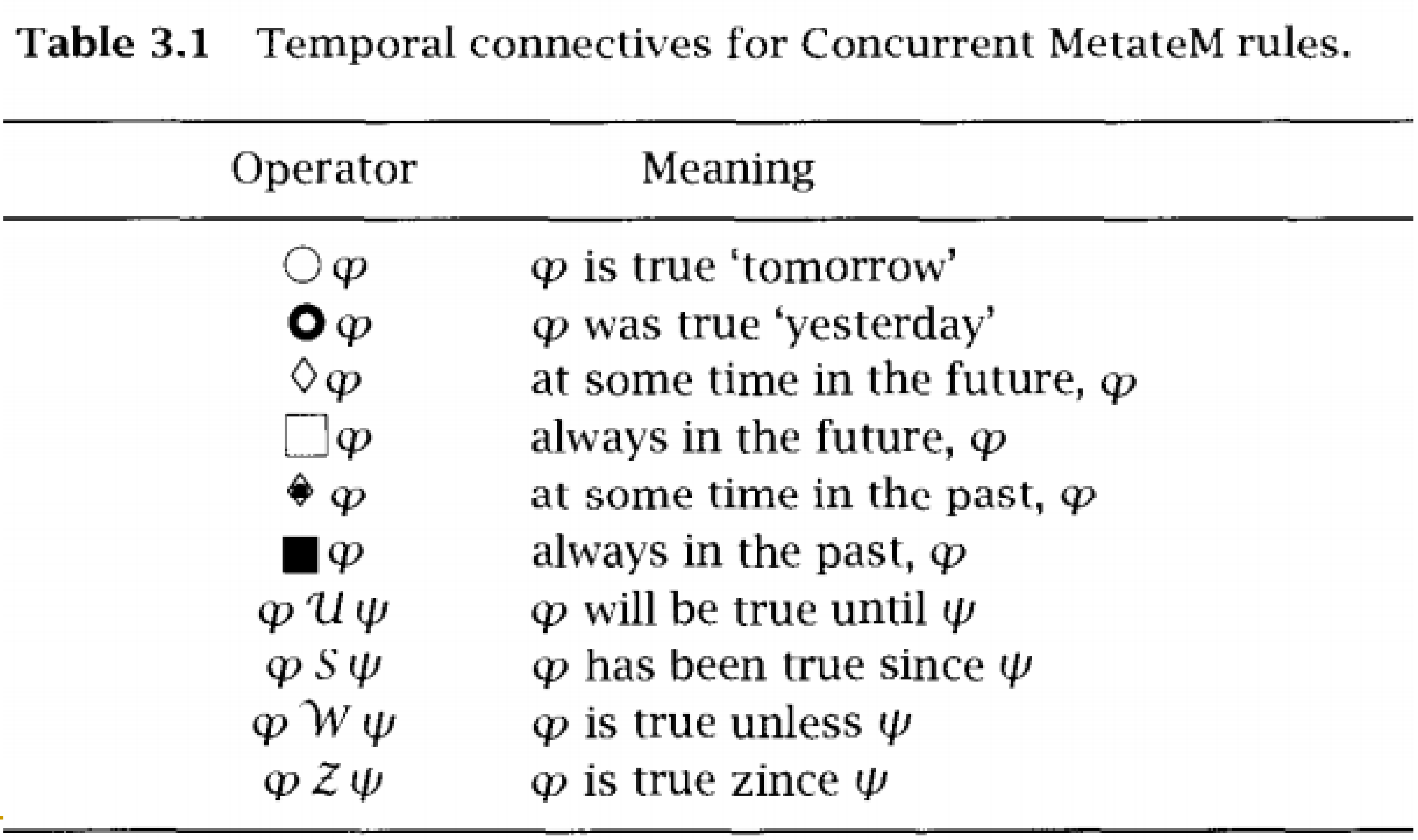


Table 3.1 Temporal connectives for Concurrent MetateM rules.

Operator	Meaning
$\bigcirc \varphi$	φ is true 'tomorrow'
$\boldsymbol{\bigcirc} \varphi$	φ was true 'yesterday'
$\Diamond \varphi$	at some time in the future, φ
$\Box \varphi$	always in the future, φ
$\blacklozenge \varphi$	at some time in the past, φ
$\blacksquare \varphi$	always in the past, φ
$\varphi \, \mathcal{U} \, \psi$	φ will be true until ψ
$\varphi \, \mathcal{S} \, \psi$	φ has been true since ψ
$\varphi \, \mathcal{W} \, \psi$	φ is true unless ψ
$\varphi \, \mathcal{Z} \, \psi$	φ is true zince ψ

Concurrent METATEM

- MetateM is a framework for *directly executing* temporal logic specifications
- The root of the MetateM concept is Gabbay's *separation theorem*:
  Any arbitrary temporal logic formula can be rewritten in a logically equivalent *past* $\Rightarrow$ *future* form.
- This *past* $\Rightarrow$ *future* form can be used as *execution rules*
- A MetateM program is a set of such rules
- Execution proceeds by a process of continually matching rules against a "history", and *firing* those rules whose antecedents are satisfied
- The instantiated future-time consequents become *commitments* which must subsequently be satisfied

Concurrent METATEM

- Execution is thus **<u>a process of iteratively</u>** generating a model for the formula made up of the program rules
- The future-time parts of instantiated rules represent *constraints* on this model
- An example MetateM program: the resource controller…

$$\forall x \quad \odot\, ask(x) \Rightarrow \Diamond\, give(x)$$
$$\forall x,y \quad give(x) \wedge give(y) \Rightarrow (x=y)$$

- First rule ensure that an ‘ask’ is eventually followed by a ‘give’
- Second rule ensures that only one ‘give’ is ever performed at any one time
- There are **<u>algorithms</u>** for executing MetateM programs that appear to give reasonable performance
- There is also *separated normal form*

Concurrent METATEM

- ConcurrentMetateM provides an **operational framework** through which societies of MetateM processes can operate and communicate
- It is based on a new model for concurrency in executable logics: the notion of executing a logical specification to generate individual agent behavior
- A ConcurrentMetateM system contains a number of agents (objects), each object has 3 attributes:
  - a name
  - an interface
  - a MetateM program

Concurrent METATEM

- An object’s interface contains two sets:
  - environment predicates — these correspond to messages the object will accept
  - component predicates — correspond to messages the object may send
- For example, a ‘stack’ object’s interface:

  stack(pop, push)[popped, stackfull]

  {pop, push} = environment preds
  {popped, stackfull} = component preds
- If an agent receives a message headed by an environment predicate, it accepts it
- If an object satisfies a commitment corresponding to a component predicate, it broadcasts it

The actual execution of an agent in Concurrent MetateM is, superficially at least, very simple to understand. Each agent obeys a cycle of trying to match the past-time antecedents of its rules against a *history*, and executing the consequents of those rules that 'fire'. More precisely, the computational engine for an agent continually executes the following cycle.

(1) Update the *history* of the agent by receiving messages (i.e. environment propositions) from other agents and adding them to its history.

(2) Check which rules *fire*, by comparing past-time antecedents of each rule against the current history to see which are satisfied.

(3) *Jointly execute* the fired rules together with any commitments carried over from previous cycles.

This involves first collecting together consequents of newly fired rules with old commitments – these become the *current constraints*. Now attempt to create the next state while satisfying these constraints. As the current constraints are represented by a disjunctive formula, the agent will have to choose between a number of execution possibilities.

Note that it may not be possible to satisfy *all* the relevant commitments on the current cycle, in which case unsatisfied commitments are carried over to the next cycle.

(4) Goto (1).

Concurrent METATEM

- Summary:
  - an(other) experimental language
  - very nice underlying theory…
  - …but unfortunately, lacks many desirable features — could not be used in current state to implement ‘full’ system
  - **currently** prototype only, full version on the way!