Reasoning with cognitive concepts

Formalization of agent systems can be used for two distinct purposes:

• as internal specification languages to be used by the agent in its reasoning;
• as external metalanguages to be used by the designer to specify, design, and verify certain behavioral properties of agents situated in a dynamic environment.

In the first use, the agent needs methods such as *theorem proving* to decide its actions.

In the second use, the designer needs methods such as *theorem proving* or *model checking* to relate logical specifications to the implementation of agents.

Temporal logics and modal logics of knowledge have been studied for some time, and theorem proving and model checking techniques have been developed for them.

Tableau-based decision procedures have been developed for some of the BDI logics.

However, formal reasoning requires much time, whereas situated agents have only limited resources for reasoning, and they must be able to react to events coming from the environment.
Implementing BDI agents

Usually the implemented BDI systems represent the BDI attitudes as data structures, rather than as modal operators.

The agent maintains some explicit representation of its BDI attitudes, as a set of beliefs $B$, a set of desires $D$, and a set of intentions $I$.

The agent receives information from the environment as percepts. When an agent gets a new percept $\rho$, it must revise its beliefs by adding the new information or changing some of the previous beliefs inconsistent with it. This process is modeled through a belief revision function $brf(B,\rho)$.

The agent must be able to perform Practical reasoning, consisting of the following activities:

• deciding what state of affairs it wants to achieve (deliberation)
• deciding how it is going to achieve this state of affairs (means-ends reasoning)

Means-ends reasoning is the process of deciding how to achieve an end (i.e. an intention) using the available means (i.e. the actions which can be performed). This process is also known as planning. The agent uses a function $plan(B,I)$ which determines a plan $\pi$ to achieve the intentions $I$.

Planning

In AI a planner is a system that takes as input:

• a goal that the agent wants to achieve
• the initial state
• the actions available to the agent

As output, a planning algorithm generates a linear plan, i.e. a sequence of actions taking from the initial state to a state satisfying the goal.

Besides linear plans, there are algorithms for generating partially ordered plans, hierarchical plans, conditional plans.
**STRIPS-like operators**

At the beginning of the 70s Fikes and Nilsson put forward a representation of actions, known as the "STRIPS-like representation" in honor of the first planner to have used it. In this representation a state is represented as a set of atomic formulas, where actions are described by a triple \(<\text{pre}, \text{del}, \text{add}>\), where \(\text{pre}\) is a set of atomic formulas representing the preconditions of the action, and \(\text{del}\) and \(\text{add}\) are the sets of atomic formulas which must be deleted from (respectively added to) the current state when the action is executed. Thus \(\text{del}\) and \(\text{add}\) describe the effects of the action.

For instance, in the blocks world, the action \(\text{stack}(x,y)\) for placing block \(x\) on top of block \(y\), can be described as:

\[
\text{stack}(x, y) \\
\text{pre} \ {\{\text{Clear}(y), \text{Holding}(x)\}} \\
\text{del} \ {\{\text{Clear}(y), \text{Holding}(x)\}} \\
\text{add} \ {\{\text{HandEmpty}, \text{On}(x, y)\}}
\]

A plan \(\pi\) is a sequence of actions \((\alpha_1, \ldots, \alpha_n)\) such that the preconditions of each action hold in the current state, and the next state is obtained by applying \(\text{del}\) and \(\text{add}\).

---

**Practical reasoning** consisting of the following activities:

- deciding **what** state of affairs the agent wants to achieve (**deliberation**)
- deciding **how** it is going to achieve this state of affairs (**means-ends reasoning**)

The deliberation phase can be decomposed into two distinct phases:

- **option generation**, in which the agent generates, via a function \(\text{options}\), a set of possible alternatives (**desires**), given the current beliefs and intentions.
- **filtering**, in which the agent chooses, via a function \(\text{filter}\), between competing alternatives, and commits to achieving them (**intentions**).
Control loop of the agent

The overall control structure of the agent is given by the following loop:

\[
\text{initialize the state;}
\]

while true do

get next percept \( p \);

\( B := \text{bfr}(B, p) \);

\( D := \text{options}(B, I) \);

\( I := \text{filter}(B, D) \);

\( \pi := \text{plan}(B, I) \);

execute(\( \pi \))

Plans

Note that there is nothing in the definition of the \textit{plan} function which requires an agent to engage in the generation of a plan from scratch, as in standard planning in AI.

In most BDI systems, the \textit{plan} function is implemented by giving the agent a \textit{library of plans}, which have been predefined by the designer of the agent. Finding a plan to achieve an intention means extracting from the library a plan that, when executed, will have the intention has a post-condition, and will be sound given the agent's current beliefs.

Our agent is committed both to ends (intentions) and to means (the plan), and never stops to reconsider its intentions or its plans.

The control loop can be refined in various ways:

- \textit{replan} if ever a plan goes wrong, i.e. it is not sound given the current beliefs;
- \textit{reconsider intentions} if they have succeeded or if they are impossible to achieve (single-minded commitment)
Refined control loop

initialize the state;
while true do
  get next percept \( \rho \);
  \( B := \text{brf}(B, \rho) \);
  \( D := \text{options}(B, I) \);
  \( I := \text{filter}(B, D, I) \);
  \( \pi := \text{plan}(B, I) \);
  while not (empty(\pi) or succeeded(I, B) or impossible(I, B)) do
    \( \alpha := \text{hd}(\pi) \);
    execute(\alpha);
    \( \pi := \text{tail}(\pi) \);
    get next percept \( \rho \);
    \( B := \text{brf}(B, \rho) \);
    if not sound(\pi, I, B) then
      \( \pi := \text{plan}(B, I) \);

Reconsidering intentions

Our agent reconsiders its intentions in the outer loop, i.e. when:
- it has completely executed a plan to achieve its current intentions, or
- it believes it has achieved its current intentions, or
- it believes its current intentions are no longer possible.

There are cases where the agent should reconsider its intentions more often. For instance, while the agent is executing a long plan to achieve some goal, it reaches a state where the same goal can be achieved in a simpler way.

Possible solution: reconsider intentions, by executing \text{options} and \text{filter}, in the inner loop as well.

We have the following dilemma:
- an agent that does not stop to reconsider its intentions sufficiently often will continue attempting to achieve its intentions even if there is no longer any reason for achieving them;
- an agent that reconsider its intentions too often may spend insufficient time actually working to achieve them.

Solution: incorporate an explicit \text{meta-level control} component, that decides whether or not to reconsider.
initialize the state;
while true do
  get next percept \( \rho \);
  \( B := \text{brf}(B, \rho) \);
  \( D := \text{options}(B, I) \);
  \( I := \text{filter}(B, D, I) \);
  \( \pi := \text{plan}(B, I) \);
  while not (empty(\( \pi \)) or succeeded(\( I, B \)) or impossible(\( I, B \))) do
    \( q := \text{hd}(\( q \)) \);
    execute(\( q \)));
    \( B := \text{brf}(B, \rho) \);
  if reconsider(\( I, B \)) then
    \( D := \text{options}(B, I) \);
    \( I := \text{filter}(B, D, I) \);
  if not sound(\( \pi \),\( I, B \)) then
    \( \pi := \text{plan}(B, I) \);

The Procedural Reasoning System (PRS), developed by Georgeff and colleagues, was one of the first agent architectures to embody the BDI paradigm.

This architecture has been quite successful, and has been re-implemented several times (DMARS, Jam, Jack, …).

It has been applied in several significant multi-agent applications, such as air-traffic control, diagnostic systems for process control applications, …

In the following we will outline the main features of PRS.
The inputs to the system are events, received via an event queue. Events can be
• external, percepts from the environment
• internal, such as addition or deletion of beliefs or goals
The outputs are external or internal actions, which are executed by the interpreter.

The system operates on explicit beliefs and goals, represented as ground literals, which can change over time.

---

**Plans**

Some of the components of a plan are:
• name
• invocation condition: a triggering event
• preconditions: the conditions that must hold before starting the execution of the plan
• body: is a directed acyclic graph whose edges are labeled with simple plan expressions. A simple plan expression is either an atomic action or a subgoal.

---

**Example**

Two plans to quench thirst.

**Name:** drink-soda  
**Invoc:** ACHIEVE quenched-thirst  
**Precond:** have-glass  
**Body:**

1. ACHIEVE have-soda
2. drink
3. GOAL

**Name:** drink-water  
**Invoc:** ACHIEVE quenched-thirst  
**Precond:** have-glass  
**Body:**

1. open-tap
2. drink
3. drink

*drink and open-tap are actions.*
Example (cont.)

A plan to get soda

Name: get-soda
Invoc: ACHIEVE have-soda
Precond: true
Body:
1. open-fridge
2. get-soda
3.  

To achieve a given end (goal), the agent forms an intention towards a means for this end, i.e. selects an applicable plan triggered by the goal. This plan becomes an intention, and is added to the intention structure.

At each step of the main loop, the interpreter selects one of the (partially executed) plans in the intention structure, and executes one step of it.

If many options are available, the interpreter may choose the one with highest utility value, or enter into metalevel reasoning using metalevel plans.

Control loop of PRS

The main steps of the control loop of PRS are the following:

• update beliefs and goals according to the events in the event queue
• the changes to the goals and beliefs trigger various plans
• one or more of the applicable plans are chosen and placed on the intention structure
• select an intention (task) from the intention structure
• execute one step of that task. This can result in
  ➢ execution of a primitive action, or
  ➢ establishment of a new subgoal, which is posted in the event queue.
Since the interpreter at each iteration selects an intention (task) and executes one step of it, task executions may be interleaved, as in a multithreaded system.

To keep track of this, each intention (task) is implemented, as usual, as a stack of frames, which describes an intermediate state of the execution of the task.

Example

Assume that event "ACHIEVE quenched-thirst" has just been added to the event queue. The invocation conditions of plans drink-soda and drink-water match this trigger event, and thus the option generator returns both plans (assume that have-glass holds).

Assume that the deliberator selects the drink-soda option, by creating a new intention for it. The first step of this plan is the goal "ACHIEVE have-soda". When this step is executed, the goal is posted in the event queue.

In the next cycle, the option generator selects the plan for getting soda, and its frame is added on top of the stack of the previous plan. The agent now executes action open-fridge, and discovers that the fridge contains no soda. The plan then fails, and the agent is forced to drop its intention to drink soda, and reposts the initial goal.

On the next cycle, the option to drink water is selected, and the plan is completed successfully over further cycles.

Agent-oriented programming

Shoham's paper Agent-oriented Programming proposes a new programming paradigm. The paradigm promotes a societal view of computation, in which multiple agents interact with one another. The paper presents:

- a restricted formal language for describing mental state;
- a programming language AGENT-0 for defining agents.
Mental categories

There are two basic mental categories: belief and obligation (or commitment). Decision (or choice) is treated as obligation to oneself. A further category, which is not a mental construct, is capability.

Formulas refer explicitly to time. A simple point-based temporal language is adopted. For instance

holding(robot,cup)

means that the robot is holding the cup at time $t$.

Actions are not distinguished from facts: the occurrence of an action is represented by the corresponding fact holding.

---

**Mental categories 2**

Belief

$B^t_a \phi$ : agent $a$ believes $\phi$ at time $t$.

For instance:

$B^3_a B^{10}_b$ like(a,b)$^7$ means that at time 3 agent $a$ believes that at time 10 agent $b$ will believe that at time 7 $a$ liked $b$.

Obligation

$OBL^t_{a,b} \phi$

means that at time $t$ agent $a$ is obligated, or committed, to agent $b$ about $\phi$. $\phi$ can be a fact representing an action.

---

**Mental categories 3**

Decision (choice)

Decision is defined as commitment to oneself:

$DEC^t_a \phi \equiv \neg OBL^t_{a,a} \phi$

Capability

The fact that at time $t$ agent $a$ is capable of $\phi$ is represented by

$CAN^t_a \phi$

Example: $CAN^5_{robot}$ open(door)$^8$
AGENT-0
A program in AGENT-0 refers to a single agent, whose name is implicit in all statements. The main syntactic constructs are:

**Facts:** \((t \text{ atom})\) atom holds at time \(t\)

**Private actions:** (DO \(t\) \(p\)-action)

**Communicative actions:**
- (INFORM \(t\) \(a\) fact) at time \(t\) inform agent \(a\) that fact
- (REQUEST \(t\) \(a\) action)

Example: (REQUEST \(1\) \(a\)) (REQUEST \(5\) \(b\)) (INFORM \(10\) \(c\) fact))

---

**Mental conditions:** a mental condition is a logical combination of mental patterns. A mental pattern is one of:
- \((B\ \text{fact})\) or \(((CMT\ a)\ \text{action})\)

CMT means "committed". The time is implicit.

**Conditional actions:** (IF mental-condition action)
- (IF \((B\ (t'\ \text{employee smith acme}))\))
- (INFORM \(t\) \(a\) \((t'\ \text{employee smith acme}))\))

---

**AGENT-0 commitments**
The conditions under which a commitment is made include both mental conditions and message conditions. A message condition is a logical combination of message patterns.

**Message pattern:** (From Type Content)
where From is the sender's name, Type is the type of the communicative action (INFORM, …) and Content is a fact or an action, depending on the type.

**Commitment rule:**
- (COMMIT messagecond mentalcond (agent action)*)
AGENT-0 commitments

Example: Flight reservation.

(COMMIT
  (?cust REQUEST (issue_bp ?pass ?flight ?time))
  (AND (? ?time (remaining_seats ?flight ?n))
    (?n>0)
    (NOT ((CMT ?anyone)
      (issue_bp ?pass ?anyflight ?time))))
  (myself (DO (+ now 1)
               (update_seats ?time ?flight –1)))
  (?cust (issue_bp ?pass ?flight ?time)))

AGENT-0 interpreter

Beliefs, commitments, and capabilities of an agent are each represented by a database. Beliefs and commitments can change at each step of the execution of a program, whereas capabilities are fixed.

The interpreter executes the following basic loop:

- read the current messages, and update your beliefs and commitments;
- execute the commitments for the current time, possibly resulting in further belief change.

Beliefs are updated either as a result of being informed, or as a result of taking a private action.

Commitments can be added as follows:

For each program statement
(COMMIT messagecond mentalcond (agenti actioni)*), if

- the message condition holds of the new incoming message;
- the mental condition holds of the current mental state;
- for all i, the agent is capable of the actioni, ... then, for all i, commit to agenti to perform actioni.
Extensions

PLACA (Thomas) extends AGENT-0 with a mechanism for flexible management of plans.

The mental state consists of beliefs, capabilities, intentions and plans.

A program consists of a set of mental-change rules:

\(<\text{message-cond}, \text{mental-cond}, \text{mental-changes}, \text{message-list}>\)

Mental changes can ADOPT or DROP believes or intentions.

Plans are maintained and generated by a separate planning mechanism that has access to the mental state, which may be regarded as a black box.

Formal methods in Agent Oriented Software Engineering

Formal methods (Wooldridge and Ciancarini) play three roles:

- in the specification of systems
- for directly programming systems
- in the verification of systems

We have already presented some logical approaches for modeling agents, such as Cohen-Levesque or Rao-Georgeff. These theories are capable of representing mental states of agents, such as beliefs, goals, intentions, actions, … Most of these approaches are based on temporal multimodal logic.

Agents can be specified in these logics, although this is not easy, due to the complexity of the underlying logics.

Formal methods in implementation

Formal specifications should provide guidelines for actually implementing agent systems. We would like to show that the implementation is correct with respect to the specifications.

This is difficult to achieve because of the large gap between specifications and standard programming languages.

A possibility:

- directly execute or animate the abstract specification

This requires to use as specification language a logic language which can be executed (computational logic)
Other approaches:

*manually refine the specification* into an executable form via some principled but informal refinement process. Methodologies for analysis and design of agent-based systems have been proposed, which can be considered as an evolution of software engineering methodologies.

*Translate or compile the specifications* into a concrete computational form using an automatic translation technique. The best-known example is the situated automata paradigm of Rosenschein and Kaelbling, who developed a technique for compiling specifications to a finite state machine.

In the following we will review some logic-based specification languages which can be directly executed. Most of these languages suffer from significant limitations, such as low efficiency, but can be used anyway for rapid prototyping.

<table>
<thead>
<tr>
<th>Abstract specifications</th>
<th>Executable specifications</th>
<th>Implementation</th>
</tr>
</thead>
</table>

**AgentSpeak(L)**

The language AgentSpeak(L) was proposed as an abstraction of one of the implemented BDI systems (in particular PRS), and allows agent programs to be written and interpreted in a manner similar to that of Horn-clause logic programs. This language can be seen as an attempt at formalizing BDI languages, by providing an operational semantics.

A different approach has been proposed by d'Inverno et al.: a version of PRS, called dMARS, has been specified by means of the specification language Z.
AgentSpeak(L) programs

Beliefs are ground atoms:

- location(robot,a), adjacent(a,b), ...

Goals can be:

- achievement goals: !location(robot,b)
- test goals: ?location(robot,b)

Triggering events can be addition/deletion of belief/goals:

+location(car,a), -location(car,b), +!location(robot,b)

Plans have the form:

+[!location(robot,X) : location(robot,Y) &
  (not X=Y) &
  adjacent(Y,Z) &
  (not location(car,Z))]

<- move(Y,Z);

+!location(robot,X).

triggering event

preconditions

body

An agent consists of a set of beliefs B, a set of plans P, a set of events E, a set of actions A, a set of intentions I, and three selection functions SE, SO and SI.

The selection function SE selects an event to process from the set of events E.

The selection function SO chooses an applicable plan triggered by the selected event.

The selection function SI selects an intention to execute.

The paper presents an operational semantics based on labeled transition systems.
GOLOG

GOLOG is a programming language based on a logic for actions (situation calculus). The interpreter of the language maintains an explicit model of the environment, which can be queried and reasoned with at run time. The language has been used for programming high-level robot control (cognitive robotics) and intelligent software agents.

Primitive actions in GOLOG are specified by giving their preconditions and effects (represented as successor state axioms).

Example:

Preconditions

\[ \text{Poss}(\text{pickup}(x), s) = \neg \text{holding}(x, s) \land \text{nexto}(x, s) \land \neg \text{heavy}(x) \]

Successor state axioms (one for each fluent)

\[ \text{Poss}(a, s) \implies \left[ \text{broken}(x, \text{do}(a, s)) = \exists r \{ a = \text{drop}(r, x) \land \text{fragile}(x, s) \lor \right. \]
\[ \left. \text{broken}(x, s) \land \neg \exists r \{ a = \text{repair}(r, x) \} \right] \]

GOLOG: complex actions

GOLOG allows to define complex actions, using the abbreviation \( \text{Do}(\delta, s, s') \) where \( \delta \) is a complex action expression; intuitively \( \text{Do}(\delta, s, s') \) will hold whenever the situation \( s' \) is a terminating situation of an execution of \( \delta \) starting in situation \( s \).

Complex actions are:

**Primitive actions:**

\[ \text{Do}(a, s, s') =_{df} \text{Poss}(a, s) \land s' = \text{do}(a, s) \]

**Test actions:**

\[ \text{Do}(\phi', s, s') =_{df} \phi' \land s = s' \]
GOLOG: complex actions

Sequence:
\[ \text{Do}(\delta_1; \delta_2), s, s') =_{df} \exists s''. \text{Do}(\delta_1, s, s'') \land \text{Do}(\delta_2, s'', s') \]

Nondeterministic choice:
\[ \text{Do}(\delta_1 \parallel \delta_2), s, s') =_{df} \text{Do}(\delta_1, s, s') \lor \text{Do}(\delta_2, s, s') \]

Nondeterministic choice of action:
\[ \text{Do}(\pi(x)\delta(x)), s, s') =_{df} \exists x. \text{Do}(\delta(x), s, s') \]

GOLOG: complex actions

Nondeterministic iteration:
\[ \text{Do}(\delta^*, s, s') =_{df} \forall P. \{ \forall s_1. P(s_1, s_1) \land \forall s_1, s_2, s_3. [P(s_1, s_2) \land \text{Do}(\delta, s_2, s_3) \Rightarrow P(s_1, s_3)] \Rightarrow P(s, s') \} \]

This is a second order definition (transitive closure is not first order definable)

It is also possible to define recursive procedures, whose semantics is given as least fixed-point.

GOLOG

This formalization of complex actions draws considerably from dynamic logic. It reifies as situations in the object language of the situation calculus the possible worlds with which the semantics of dynamic logic is defined.

GOLOG (aKol in LOGic) is designed as a logic programming language for dynamic domains. It attempts to blend ALGOL programming style into logic. It borrows from ALGOL many well-known programming constructs such as sequence, conditionals, recursive procedures and loops. For instance:

\[
\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2 =_{df} [\phi; \delta_1] \upharpoonright [\neg \phi; \delta_2]
\]
**GOLOG: an elevator controller**

**Primitive actions:** up(n), down(n), open, ...

**Fluents:** current_floor(s) = n, next_floor(n, s), ....

**Preconditions:** .........

**Successor state axioms:** ..............

**Procedures:**

```prolog
proc control [while ∃n. on(n) do serve_a_floor endWhile];
  park endProc
proc serve_a_floor (π n) [next_floor(n)?; serve(n)] endProc
proc serve(n) ......................
```

---

**GOLOG: running a program**

Executing a program amounts to establish the following entailment

\[ \text{Axioms} \models \exists s. \text{Do(program, } S_0, s) \]

where \( S_0 \) is the initial situation.

A successful execution of the program, i.e. a successful proof, returns a binding for \( s \):

\[ s = \text{do}(a_2, \ldots, \text{do}(a_1, S_0)\ldots) \]

- GOLOG’s interpreter is a general-purpose theorem prover (in general for second order logic).
- Like Prolog, GOLOG programs are executed for their side effects, i.e. to obtain bindings for existentially quantified variables.

---

Various extensions of GOLOG have been developed:

CongoLOG incorporates concurrency, handling concurrent processes with different priorities, high-level interrupts, exogenous actions.

IndiGolog: programs can be executed incrementally to allow for interleaved action, planning, sensing, and exogenous events.

A related approach is that of DyLog (Torino), which uses a modal logic of actions instead of situation calculus. In particular, procedures can be defined by means of Prolog-like rewrite rules, which are interpreted as axioms of the logic (grammar logics).
Concurrent METATEM

Concurrent METATEM (Fisher …) is a language based upon the direct execution of temporal formulae.

A Concurrent METATEM system contains a number of concurrently executing agents, each of which is able to communicate via asynchronous broadcast message passing.

Each agent is programmed by giving a temporal logic specification of its behavior. This specification can be executed directly.

The linear logic used by Concurrent METATEM is based on a discrete, linear model of time, with finite past and infinite future.

The time is modeled as an infinite sequence of discrete states, with an identified starting point. Formulae are interpreted at particular states, and the language provides operators which refer to both the past and future.

Some operators are:

**future:**
- \( \Box \varphi \) \( \varphi \) must be satisfied in the next state
- \( \varphi \lor \psi \) \( \varphi \) will be true until \( \psi \)
- \( \Diamond \varphi \) \( \varphi \) must be satisfied at some state in the future
- \( \Diamond \varphi \) \( \varphi \) must be satisfied at all states in the future

**past:**
- \( \varphi S \psi \) \( \varphi \) has been true since \( \psi \)
- \( \Diamond \varphi \) \( \varphi \) was satisfied in the previous state

A METATEM program consists of a set of rules of the form:

\( \Box (\text{past and present formula} \Rightarrow \text{present or future formula}) \)

Where the right-hand side is, roughly, constrained to be either a disjunction or a sometime formula.

The language provides two orthogonal mechanisms for representing choice:

- static indeterminacy, through the classical operator \( \lor \)
- temporal indeterminacy, through the sometime operator \( \Diamond \) (however, given \( \Diamond \varphi \), the execution mechanism attempts to satisfy \( \varphi \) as soon as possible).
Example (☐ is omitted):  
\[ \text{start} \Rightarrow \text{popped(a)} \]

\text{popped(a)} is satisfied at the beginning of time (\text{start} is a nullary operator which is true at the beginning of time)

\[ \text{pop}(X) \Rightarrow ☐\text{popped}(X) \]

whenever \text{pop}(X) is satisfied in the previous moment of time, a commitment to eventually satisfy \text{popped}(X) is given (X is a universally quantified variable)

\[ \text{push}(X) \Rightarrow \text{stack-full} \lor \text{popped}(X) \]

whenever \text{push}(X) is satisfied in the previous moment of time, then either \text{stack-full} or \text{popped}(X) must be satisfied.

The interpreter continually executes the following step:
check which rules have antecedents satisfied,
conjoin together the consequents of these rules,
rewrite this conjunction into a disjunctive form and
choose one of these disjuncts to execute.
If a contradiction is found, it may be possible to backtrack to a previous choice (however backtracking is not allowed after broadcasting of a message)

The basic mechanism of communication among agents is broadcast message-passing.

Three categories of predicates:

- \text{Environment predicates}: represent incoming messages. For instance \text{push}(X) is true if a message push(b) has just been received.
- \text{Component predicates}: represent messages broadcast from the agent. For instance, when \text{popped}(c) is made true, then the message \text{popped}(c) is broadcast.
- \text{Internal predicates}. 

Any agent has an **interface** which defines how the agent may interact with the environment.

An interface consists of:
- an agent identifier
- a list of environment predicates (the messages the agent recognizes)
- a list of component predicates (the messages that the agent may send)

For instance:

```
stack (pop, push) [popped, stack-full]
```

Example. A 'resource producer':

```
rp (ask1, ask2) [give1, give2]
\[\text{O} \text{ask1} \Rightarrow \neg \text{give1} \quad \text{commits to eventually give to any agent that asks}
\text{O} \text{ask2} \Rightarrow \neg \text{give2}
\text{start} \Rightarrow \square (\neg \text{give1} \land \neg \text{give2}) \quad \text{can give to only one agent at a time}
```

Two consumers:

```
rcl (give1) [ask1]
\text{start} \Rightarrow \text{ask1}
\text{O} \text{ask1} \Rightarrow \text{ask1} \quad \text{ask on every cycle}
rc2 (ask1, give2) [ask2]
\text{O} (\text{ask1} \land \neg \text{ask2}) \Rightarrow \text{ask2} \quad \text{an ask2 message is sent on every cycle where, on its previous cycle, it did not send ask2 but received ask1.}
```

### 3APL

3APL (Hindriks …) incorporates features from both imperative and logic programming.

Its main features are:
- representing and querying the agent's beliefs,
- belief updating, for incorporating new and removing existing information in the agent's belief base,
- goal updating, to facilitate practical reasoning, that is, for planning and the reconsideration of adopted plans.

An intelligent agent consists of a **goal base**, a **belief base**, and a set of practical reasoning rules.
Beliefs: are formulas in some logical language.

Goals can be:
• a basic action,
• an atom: achievement goal,
• a test goal ϕ?
• a complex goal
  • sequential composition: π₁; π₂
  • nondeterministic choice: π₁ + π₂

Basic actions may update or change the beliefs of an agent.

Practical reasoning rules have the form:

πₐ ← ϕ | πₖ

where πₐ and πₖ are (possibly complex) goals.
Informally, one can read it as stating that if the agent has adopted some goal π which matches with πₐ, and believes ϕ, then it may consider adopting goal πₖ as a new goal.
Practical reasoning rules can be use to build a plan library from which an agent can retrieve plans for achieving an achievement goal.
A rule with an empty body πₖ can be used to drop a goal.
A rule with an empty head πₐ can be used to create a new goal.

PROGRAM "cleaning"
CAPABILITIES:
{pos(P)} Goto(R) {NOT pos(P), pos(R)},
{pos(P) AND dirty(R)} Vacuum(R) {NOT dirty(R)},
{pos(P1) AND box(P2)} MoveBox(P1,P2) {NOT pos(P1) AND NOT box(P1) AND pos(P2) AND box(P2)}
BELIEFBASE:
dirty(room1), dest(room1), box(room2), pos(room3)
GOALBASE:
clean(), transport()
RULEBASE:
clean() <- dirty(Room) |
BEGIN
  Goto(Room);
  Vacuum(Room)
END.
transport() <- box(Room) AND dest(Dest) |
BEGIN
  Goto(Room);
  MoveBox(Room,Dest)
END.
The rules can be: failure rules, reactive rules, plan rules, and optimization rules, with different priorities.

The semantics of 3APL is given by means of a transition system.

The language has been extended with a meta-language that provides programming constructs to implement the deliberation cycle of the agent.

**IMPACT**

The IMPACT system has been developed by Subrahmanian et al. to support the creation and deployment of multiple software agents, which can interoperate with a wide variety of custom-made, as well as legacy software sources.

An IMPACT agent consists of two parts:

1. A body of software code (built in any programming language) that support a well defined application programming interface (API). This software consists of a set of data (representing the state of the agent) with a set of operations on them (e.g. a database with the usual operations)

2. A semantic wrapper containing various semantic information (service description, message manager, …)

An agent program specifies what actions an agent is obliged to take in a given state, what actions it is permitted to take, and how it chooses which actions to perform.

Various applications have been developed with IMPACT:
- a supply chain management system,
- a multagent solution to the "controlled flight into terrain" problem with a flight planning agent, terrain elevation agents, and GPS agents.
Software code access in IMPACT

Given a software code $S$, a code call is $S:f(d_1, \ldots, d_n)$, where $f$ is a function (operation) on the data in $S$.

- oracle:select(emp.rel, salary, $>$, 150000)
executes a select operation on the emp.rel table and returns the set of all tuples whose salary is over 150000.

A code call atom is an expression $\text{in}(t, cc)$, where $t$ is a term and $cc$ is a code call.

- $\text{in}(X, \text{oracle:select(emp.rel, salary, $>$, 150000)})$
variable $X$ is instantiated to a tuple whose salary is over 150000.

A code call condition is a conjunction of code call atoms.

Integrity constraints specify properties that states of the agent must satisfy:

- $\text{in}(X, \text{oracle:select(emp.rel, salary, $>$, 150000)}) \Rightarrow X.\text{grade} > 5$

Actions

Agents can execute actions. Every action $\alpha$ has a precondition $\text{Pre}(\alpha)$, and a set of effects (given by an add list $\text{Add}(\alpha)$ and a delete list $\text{Del}(\alpha)$) that describe how the agent state changes when the action is executed.

Action status atoms, make use of deontic logic operators:

- $P\alpha$ the agent is permitted to take action $\alpha$
- $F\alpha$ the agent is forbidden from taking action $\alpha$
- $O\alpha$ the agent is obliged to take action $\alpha$
- $W\alpha$ the obligation to take action $\alpha$ is waived
- $D\alpha$ the agent does take action $\alpha$

An action rule is a clause of the form

- $A \leftarrow L_1, \ldots, L_n$
where $A$ is an action status atom, and each $L_i$ is either an action status atom or a code call atom, possibly negated.

An agent program is a collection of rules.
Example: Tax audit agent

The agent determines which users should be audited by monitoring two relations: *returns*, containing the returns filed by tax-payers *employer_decs*, specifying the payments reported by employers.

If the amount reported by an individual is less than 70% of the total income of that individual reported by all employers, then triggering an audit program is mandatory. If the reported amount is less than 80% of the total income reported by all employers, then triggering an audit program is permitted. However, if the amount reported by the individual is over 80% then it is forbidden to run the audit program.

The program is:

\[
\begin{align*}
O(\text{run_audit(Person)}) & \leftarrow \text{in}(R, \text{taxdb:select(returns, name, =, Person)}), \\
& \text{in}(\text{TotalInc}, \text{taxdb:sum_employer_decs(Person)}), \\
& R.\text{amount} \leq 0.7 \times \text{TotalInc}.
\end{align*}
\]

\[
\begin{align*}
F(\text{run_audit(Person)}) & \leftarrow \text{in}(R, \text{taxdb:select(returns, name, =, Person)}), \\
& \text{in}(\text{TotalInc}, \text{taxdb:sum_employer_decs(Person)}), \\
& R.\text{amount} \geq 0.8 \times \text{TotalInc}.
\end{align*}
\]

\[
\begin{align*}
P(\text{run_audit(Person)}) & \leftarrow \text{in}(R, \text{taxdb:select(returns, name, =, Person)}), \\
& \text{in}(\text{TotalInc}, \text{taxdb:sum_employer_decs(Person)}), \\
& R.\text{amount} \leq 0.8 \times \text{TotalInc}.
\end{align*}
\]

where \( \text{sum}_\text{employer_decs(Person)} \) is a query provided by the taxdb which returns the sum of all the payments to *Person*.

Semantics for agent programs

The semantics must answer the question: What is the set of all action status atoms of the form \( Do \alpha \) that are true in a given state with respect to the program and to the underlying constraints?

To do this, the notions of feasible and rational status set are introduced, where a status set is a set of ground action status atoms.

For instance, a feasible status set \( S \) is

\begin{align*}
deontically \text{ consistent: if } O\alpha \in S, \text{ then } W\alpha \notin S \quad & \ldots \\
\text{and} & \\
deontically \text{ closed: if } O\alpha \in S, \text{ then } P\alpha \notin S \quad & \ldots
\end{align*}

Intuitively, a feasible status set consists of assertions about the state which are compatible with the program of the agent and the underlying constraints.

For instance, assume that John Smith has declared under 70% of his income, Jane Shady between 70 and 80%, and Denis Rumble more than 80%. Then, two possible feasible status sets are:

\[ FSS_1 = \{ \text{O run\_audit(John Smith), P run\_audit(John Smith),} \]
\[ \text{Do run\_audit(John Smith), F run\_audit(Denis Rumble),} \]
\[ \text{P run\_audit(Jane Shady)} \} \]

\[ FSS_2 = \{ \text{O run\_audit(John Smith), P run\_audit(John Smith),} \]
\[ \text{Do run\_audit(John Smith), F run\_audit(Denis Rumble),} \]
\[ \text{P run\_audit(Jane Shady), Do run\_audit(Jane Shady)} \} \]