

AF-APL – Bridging Principles & Practice in Agent Oriented Languages

Robert Ross
Universitat Bremen
Germany
robertr@tzi.de

Rem Collier
University College Dublin
Ireland
rem.collier@ucd.ie

G.M.P. O’Hare
University College Dublin
Ireland
gregory.ohare@ucd.ie

Abstract

For AOP (Agent Oriented Programming) to become a mature discipline, lessons must be learned from practical language implementations. We present AF-APL (AgentFactory - Agent Programming Language) as an Agent Oriented Programming Language that has matured with continued revisions and implementations, resulting in a language - which, although based on the more theoretical aspects of AO design - has incorporated many of the practical considerations of programming real world agents. We describe AF-APL informally, focusing on its experience driven features, such as commitment reasoning, a rich plan operator set, and an inherent asynchronous design. We present the default execution cycle for the AF-APL interpreter, looking in detail at the Commitment Management model. This model provides an agent with power to reason about its own actions, while maintaining basic constraints on computational tractability. In our development of the language, we learned many lessons that are not covered in the purer AO language definitions. Before concluding, we discuss a number of these lessons.

1. Introduction

Agent-Oriented Programming (AOP) represents one approach to the fabrication of agent-oriented applications. It is based upon the premise that complex agent behaviors can be best represented in terms of a set of mental notions (e.g. belief, goal, and commitment), and the interplay between them.

Research in this area has produced a number of agent programming languages, including Agent0 [17], 3APL [5], AgentSpeak(L) [13], and AF-APL [3]. Many of these languages draw upon earlier work on logical agent theories that use Possible Worlds semantics [1] [2] [9]. However, computation tractability issues have led to a number of them being re-specified using more tractable approaches such as formal methods (e.g. VDM and Z) and operational semantics.

While the use of formal semantics has resulted in AOP languages that are conceptually well defined, many of them have not been applied to large scale projects. These projects raise a number of practical issues regarding the use of AOP languages, such as usability, scalability, and applicability.

It is these issues that have driven the work presented in this paper. Specifically, we describe an extended version of AF-APL [3], an AOP language that was originally derived from a logical model of commitment. These extensions, are rooted in practical experiences gained during the last 3 years. Specifically, AF-APL has been employed in the development of a number of large scale agent-oriented applications in the robotics [16] and mobile computing [11] domains. These experiences have driven a number of extensions to AF-APL that include: the introduction of goals, additional plan operators, a new commitment management system, support for commitment reasoning, and reactive rules.

The rest of the paper is structured as follows: Section 2 presents a review of significant work to date in AO language development. This is followed in Sections 3 and 4 with a description of AF-APL - the first of these sections details AF-APL’s constructs, while Section 4 looks at the AF-APL execution model. Although this paper focuses on the AF-APL language, we have presented a basic overview of an AF-APL interpreter; That interpreter, based on the AgentFactory Framework, is presented in Section 5. Section 6 then illustrates the use of the language, with a toy example from the mobile robot domain. In our development of the language, we learned many lessons that are not covered in the purer AO language definitions - before concluding, Section 7 discusses a number of these lessons.

2. Related Work

A number of deliberative agent languages and architectures are already present in the literature. While some of these explore agent language theories, others have focused on the provision of complete agent frameworks.

Inspired by Dennett’s Intentional Stance [7], Shoham’s

AgentO [17] established AOP as a programming paradigm on a par with OOP. Importantly, AgentO provided a clean language that could allow designers to build programs from mental attributes such as Beliefs and Commitments. Limitations of AgentO included no explicit support for Declarative Goals, a lack of plan operators, and little detail of how the high level programming language could interact with lower level code.

Rao’s AgentSpeak(L) [13], based on the Procedural Reasoning System (PRS) [14], offered a more powerful agent design; AgentSpeak(L) agents had a mental state including Beliefs, Plans, Goals, Actions, Events, and Intentions. Rao’s language is notable first as an AOP language grounded in action, and second as an AOP language that can be formally verified. However, the language intentionally left open the question of what choice functions should be used in the selection of intentions and plans. This, along with a limited set of plan operators, meant that AgentSpeak(L) was not, by itself, a suitable candidate for a deployable AO language.

A more recent AO language worthy of mention is *Goal Directed 3APL* [10]. That language, like its more abstract parent 3APL [6], strives for a separation of mental attributes and the reasoning process. This separation, facilitated with meta-language programming, leads to a well defined programming language, with a clear Operation Semantics provided by Transition Systems. 3APL provides a rich set of mental objects including Beliefs, Goals, Plans, and various Reasoning Rules. However, 3APL does not provide a fixed deliberation process. Instead, it is intended that designers be allowed to directly specify an agent’s deliberation process, as well as its capabilities and base knowledge. Also, as with AgentO, there is only a primitive relationship between a 3APL action and the manipulation or sensing of an agent’s environment.

Other recent entrants into the deliberative agent development literature include Nuin [8] and Jadex [12]. Both of these are agent architectures that aim to provide a practical framework for BDI style agents. Nuin’s underlying BDI model builds on AgentSpeak(L), addressing issues like AgentSpeak(L)’s lack of choice operators. However, like PRS and AgentSpeak(L), Nuin requires designers to define fitness functions for the selection of current attention (i.e. `select-focus`). The Jadex BDI model is more basic, including Beliefs, Plans, and *to-do* Goals (Commitments). Jadex lacks declarative goals and means-end reasoning, plus there is an unclear relationship between object oriented and agent oriented design in plan definitions. Both Nuin and Jadex directly address *middleware* concerns such as message transportation, migration, and yellow pages support. Although provision of such services is doubtlessly valuable, combining these *middleware* issues directly with agent language or architecture design potentially precludes the development of more powerful control algorithms.

We conclude that the literature includes rich, theoretical, languages with well defined semantics (3APL), along with BDI frameworks for the development of AO applications (Nuin). However, these approaches either lack explicit deliberative process description, or they are unclear in how the language would work in real world deployment. It is our belief, that for AOP to become a mainstream programming methodology, AO languages must: (1) utilise a rich BDI style feature-set; (2) have a clear deliberation process; and (3) be a core language upon which other *middleware* and application specific aspects of agent deployment can be built. In the rest of this paper we present AF-APL as a language that has matured through application experience. In doing so, we hope that some of some of our lessons learned can be valuable to others.

3. AF-APL Constructs

The main constructs of any Agent Oriented Programming Language inevitably centre around objects such as Beliefs, Goals, Commitments and Plans. We now overview AF-APL’s collection of base objects and meta constructs.

3.1. Core Logical Constructs

From the object perspective, an AF-APL agent is defined as a tuple of mental objects:

$$Agent = \{B, G, C, A, P, BR, RR, CR, PR\}$$

where B is the agent’s Belief Set, G is the agent’s Goal Set, C is the agent’s Commitment Set, A is the agent’s Actuator Set, P is the agent’s Perceptor Set, BR is the agent’s Belief Rule Set, RR is the agent’s Reactive Rule Set, CR is the agent’s Commitment Rule Set, and PR is the agent’s set of Plan Rules, or simply Plans. We now informally introduce each of the AF-APL constructs which are used to create and manipulate these objects.

3.1.1. Beliefs In AF-APL, a belief is represented with the BELIEF construct. By default a belief only persists for one iteration of the agent’s execution cycle. This behavior can be changed through the use of the NEXT, UNTIL, or ALWAYS constructs as required. We specify that an agent believes that Rem always likes beer, as follows:

```
ALWAYS(BELIEF(likes(Rem, beer)))
```

3.1.2. Perceptors An AF-APL agent contain a non-empty set of perceptors \mathcal{P} , each of which enable the agent to acquire beliefs about its environment. The PERCEPTOR construct is used to declare a perceptor with a particular identifier and external code implementation. Each perceptor is fired in its own thread once per agent execution cycle, and any resultant beliefs are then added to the agent’s belief set

3. For example, to define a Java based perceptor for receiving FIPA messages, one might use this construction:

```
BEGIN{PERCEPTOR}
IDENTIFIER ie.ucd.af.fipa.fipaReceive;
CODE ie.ucd.af.fipa.Receiver.class;
END{PERCEPTOR}
```

Here, the IDENTIFIER of the perceptor is a unique perceptor identifier within the AF-APL namespace. CODE defines a piece of *external code* that is used to implement the perception task. Perceptor implementation can be provided as Java classes or C libraries, that make use of a defined Perceptor interface. Using these interfaces Perceptors add new beliefs to the agent's belief set. Multiple CODE declarations can be defined, but only one piece of code is used on any platform. This can be useful when dealing with migrating agents over multiple hardware platforms. Take for example a mobile agent which crosses between a high performance PC, and a computationally limited handheld device. Conceivably, when the agent moves to the low-spec platform, it may need to switch to a more basic form of perceptor.

3.1.3. Actuators AF-APL agents contain a non-empty set of actuators \mathcal{A} , each of which constitutes the most basic action an agent can perform and reason about. As with perceptrs, actuators are defined in terms of a particular identifier, with an implementation provided in an external programming language. In addition, actuators have pre and post conditions defined; these conditions determining what must be true for an actuator to be invoked, and what should be true when an actuator has completed. Axioms of the language also state that if an actuator succeeds or fails, then the agent directly adopts a belief to that effect; this technique has been found to be extremely useful in allowing an agent to reason about the success of its own actions. To avoid locking of the agent execution cycle, actuators are fired asynchronously. An actuator for sending FIPA messages might be defined as follows:

```
BEGIN{ACTUATOR}
IDENTIFIER <BEHAVIOURAL>
    ie.ucd.af.fipa.fipaSend(?fipa_msg);
PRE BELIEF(TRUE);
POST BELIEF(send_success(?fipa_msg))
    | BELIEF(send_failure(?fipa_msg));
CODE ie.ucd.af.fipa.Sender.class;
END{ACTUATOR}
```

The actuator identifier (like the perceptor and plan identifiers) uses a name-spacing convention which is similar to (and compatible with) the Java namespace. The actuator identifier normally only refers to the shortened form of this term (i.e. `fipaSend(?fipa_msg)`), but the long form can be used to distinguish between two actuators with the

same short form identifiers. Actuators are by default assumed to be functional in nature (expected to self terminate); however, this interpretation can be overwritten by declaring the actuator to be behavioral by using the optional BEHAVIOUR keyword in the IDENTIFIER (See the DO_BEHAVIOUR_UNTIL construct for more details).

3.1.4. Commitments A commitment is a promise made by an AF-APL agent to attempt an action. In other words, a commitment is the mental equivalent of a contract. As such, it specifies the course of action that the agent has agreed to; to whom this agreement has been made; when it must be fulfilled; and under what conditions the agreed course of action becomes invalid (i.e. under what conditions the contract can be breached). At any time the agent can contain any number of commitments in its commitment set \mathcal{C} . For example, we can represent that Rem has committed himself to eat biscuits at 11am (as long as he believes that he has no lunch plans) with the COMMIT construct as follows:

```
COMMIT(Rem,
    11:00,
    !BELIEF(has(Rem,LunchPlans)),
    eat(biscuits))
```

The first argument taken by a commitment construct is the name of the Agent to whom this commitment has been made; this can be any literal term, or the keyword ?Self, which is replaced at runtime with the name of the AF-APL agent. The second argument is the earliest time at which the agent should attempt to realize the commitment. The start time may be specified absolutely in the standard international date format YYYY/MM/DD-hh:mm:ss, or relative to the adoption time of the commitment by affixing a '+' character to the start of a time (e.g. +01:00:00 means that the action should be started one hour after the commitment was adopted. The key term '?Now' can also be used to specify that the minimum start time for the action equals the commitment adoption time. The third term is the commitment's maintenance condition; if at any time after adopting the commitment the term is no longer entailed by the agent's belief set, then the commitment is dropped immediately. The final term is the action to be performed by the agent; The commitment holds a special place in the AF-APL language, since it is through commitment management, and commitment revision that an agent performs intentional action. Section 4.1 overviews of the commitment management process.

3.1.5. Goals A goal is a state of the world - or set of beliefs - that an agent wishes to bring about. For example, the goal of causing the door to be closed is represented as:

```
GOAL(door(closed))
```

Agents may adopt goals using either the ADOPT_GOAL or ACHIEVE_GOAL plan operators, as described in section

3.1.9. Once adopted into \mathcal{G} , an agent will use means-end reasoning to attempt to determine a plan that can achieve the goal. If such a plan can be determined, a secondary commitment to that plan structure will form as a child of the original goal; fulfilling the plan then causes the original commitment to be dropped. To guarantee continued reactive behavior of the agent, the means-end reasoning process is performed asynchronously - this behavior is discussed further in section 4.

3.1.6. Belief Rules At any time the agent has a number of Belief Rules that allow the agent to infer new beliefs from already existing beliefs. Therefore, the agent can decide if a friend is allowed to drink beer with the following Belief Rule:

```
BELIEF(friend_age(?friend,?years))
& BELIEF(greaterThan(?years,18))
=> BELIEF(canDrinkAlcohol(?friend));
```

The left hand side of a belief rule is referred to as a belief query, and is a conjunction of positive or negative current beliefs containing free or bound variables. Belief queries can reference the agent's belief set directly, or make use of three relational operations equals(?x,?y), greaterThan(?x,?y) and lessThan(?x,?y).

3.1.7. Commitment Rules As well as belief rules, an AF-APL agent holds a set of commitment rules, \mathcal{CR} , that allow the agent to rationally decide on a course of action based on its mental state. Take for example an agent who is hungry and has a piece of fruit. Under these circumstances, a rational action for the agent to take, would be to eat the fruit. We can specify this as a commitment rule in AF-APL with the following:

```
BELIEF(hungry(?Self))
& BELIEF(haveFruite(?Self,?fruit))
=> COMMIT(?Self,
    ?now,
    BELIEF(TRUE),
    eat(?fruit));
```

The left hand side of a Commitment Rule is a Mental State Query. As well as allowing querying of \mathcal{B} , a mental state query can also query \mathcal{G} and \mathcal{C} . Thus, an agent can adopt commitments based not only on its beliefs, but also on the actions it has already committed to. It is often useful to make judgements based on what the agent believes to be true in terms of commitment precedence and timing. We therefore introduced two mental functions which allow an agent to determine basic relationships between commitments; *before* allows an agent to determine if the initial start time of one commitment precedes another; while *consequenceOf* allows an agent to explicitly determine if one commitment is a direct result of another commitment (e.g. is a descendent).

3.1.8. Reactive Rules An AF-APL agent contains a set of Reactive Rules \mathcal{RR} that allow an agent to invoke an action directly based on what it believes to be true. The reactive rule is simpler than a commitment rule, but is less robust and its consequences cannot be reasoned about by the agent. A reactive rule can be used to code basic reactive behaviors into the agent's design. For example, a reactive rule to dodge an obstacle blocking a robot's progress could be encoded with:

```
BELIEF(blocked(ahead))
& BELIEF(moving(forward))
=> EXEC(dodgeObstacle);
```

Reactive rules are different from their big brothers - the commitment rule - in a number of ways, including: static binding, reactive rules are bound at runtime to a specific plan implementation, whereas choices between plans and actuators can be made by commitment management at runtime; no deliberation, the agent cannot reason about reactive responses, nor can the reactive rule trigger contain COMMIT or GOAL structures; priority execution, reactive rules take precedence over commitment rules; more basic formulation, reactive rules may only contain actuator identifiers or the SEQ, XOR, OR, PAR, and AND plan constructs.

3.1.9. Plan Operators In all the commitment rule and reactive rule examples above, only single actions were to be performed. As would be expected, we explicitly provide a number of plan operators for constructions of complex action from primitives.

- **SEQ** - The Sequence Construct defines a set of steps that must be realized in the order listed. For example, the actions required to boil a kettle can be combined as:

```
SEQ(getKettle,
    fillKettle,
    boilKettle)
```

SEQ, along with PAR, AND, OR, and XOR are collectively known as the arrangements operators. These operators can operate on any whole number of arguments. The case of operating on one argument is not particularly interesting; the operator is said to succeed if its one argument succeeds, and fails otherwise.

- **PAR** - The Parallel Construct defines a set of steps that should be realized simultaneously. We can express the parallel actions of stirring and adding milk as:

```
PAR(stir,addMilk)
```

- **AND** - The Random Order Construct defines a set of steps which can be realized in any order, parallel or sequential. All steps must be performed, but it really does not matter what order the steps are performed in. Coming back to the tea example: consider the actions of

getting tea, getting a cup, and getting the milk. There is no particular ordering necessary here; therefore we can use the AND construct as follows:

```
AND(getTea,
    getCup,
    getMilk)
```

- **OR** - The Non-Deterministic Choice Construct defines a set of steps, one of which must be realized. All steps are attempted in parallel. Once one of these steps is realized, the construct is said to have succeeded; resulting in all other steps being abandoned. If none of the steps return true, then the construct is said to have failed. Using the tea example again, it is of course possible to boil water in more than one way. In addition to using a kettle, water may also be boiled in a pot. This results in two alternative ways to get boiling water, which can be expressed in a plan body as:

```
OR(SEQ(getKettle,
        fillKettle,
        boilKettle),
    SEQ(getPot,
        fillPot,
        boilPot)
)
```

The OR construct is useful when we wish to try out many different options at the same time. However, in practice, it is not always useful to perform all operations together. In our boiling water example, it is probably a bad idea to both try to get boiling water from the pot and from the kettle at the same time. Instead, we often try each option in turn; to do this we use the XOR construct.

- **XOR** - The Deterministic Choice Construct defines a set of steps, which must be performed in the order presented, and which succeeds when one of the steps is realized. Our boiling water example can be expressed with XOR:

```
XOR(SEQ(getKettle,
        fillKettle,
        boilKettle),
    SEQ(getPot,
        fillPot,
        boilPot)
)
```

- **FOREACH** - The Universal Quantification Construct allows an agent to check the contents of its mental state, and assign variables into plans based on this mental state. FOREACH takes two arguments: a belief query sentence, and a plan body with free variables (all of which must be potentially scoped by the belief query sentence). To illustrate consider an agent

that wants to hold a party, and therefore wishes to invite all its friends to the party; we can express this with the FOREACH construct as follows:

```
FOREACH(BELIEF(friend(?name)),
        AND(invite(?name)))
```

At runtime, if the agent's belief set includes the beliefs that it has friends: Anne, Jane and Freddy, then the above statement will be expanded to:

```
AND(invite(Anne),
    invite(Jane),
    invite(Freddy))
```

The plan body to be expanded must: (a) be based on an arrangement construct e.g. XOR, OR, AND, PAR or SEQ; (b) contain only one argument. If the belief query sentence fails - in this case, because the agent has no friends - then the FOREACH construct is said to fail.

- **TEST** - The Belief Query Construct allows us to test if particular beliefs are held by the agent. The construct takes one argument, a belief query sentence, which may or may not contain free variables. If the agent's mental state entails the belief query sentence, then the construct is said to succeed. We can use TEST to decide if we want to drink the tea that we have made:

```
SEQ(tasteTea,
    XOR(SEQ(TEST(BELIEF(tea_tastes(good))),
        enjoyTea),
        SEQ(TEST(BELIEF(tea_tastes(bad))),
            drinkItAnyway),
        drinkCoke))
```

Using the TEST construct is equivalent to creating an actuator which explicitly checks the mental state of the agent; however, the construct is defined as part of the language; meaning it is used more efficiently than an external actuator.

- **DO_BEHAVIOUR_UNTIL** - The Behavior Controller Construct is used to handle actuators which have a behavioral rather than functional nature. All of the examples above presumed actuators to have a functional nature, in that they were expected to terminate eventually. Some actions, particularly in robotics, have something closer to a behavioral nature in that they do not have a natural termination point, and are only stopped once the agent believes something to be true. The DO_BEHAVIOUR_UNTIL construct takes two arguments: (a) the identifier of an actuator which has been declared BEHAVIOUR; (b) a belief query sentence, which can contain free variables. The actuator is invoked and will be left to run until it returns or the agent's belief set entails the query; in which case, the actuator is forcefully stopped. We can specify that a robot is to move along a wall until a door is found with:

```
DO_BEHAVIOUR_UNTIL( followWall,
                   BELIEF( found(Door) ) )
```

- **TRY_RECOVERY** - The Error Recovery Construct is used to indicate whether an action - or indeed a plan structure - should have error recovery mechanisms associated. The construct takes two arguments: first, the action or plan body to be monitored for failure; second, the plan which is to be used for recovery. Unlike the *try catch* exception handling mechanisms in Java, TRY_RECOVER attempts to repair an erroneous situation and return the agent to finish the original action. For example, a robotic agent that has to perform a number of movement actions, can use a social recovery plan [15] to guard against erroneous situations as follows:

```
TRY_RECOVER( SEQ( moveForward( "5m" ),
                 turn( "90d" ),
                 enterDoor ),
            social_recovery )
```

- **CHILD_COMMIT** - The Commitment Adjustment construct is used to override the default secondary commitment creation semantics of the language. As with the COMMIT construct introduced above, the CHILD_COMMIT construct takes four arguments, explicitly overriding to whom the secondary commitment is made; when the commitment is to be first attempted; under what conditions the commitment is dropped; and the action to be achieved. If only a subset of these parameters is to be overwritten, then the ?default key term can be used to indicate that the default value is to be kept. For example, if within a complete plan, there are two branches, one of which is to Jim, with another branch to Anna, then we can express this as follows:

```
CHILD_COMMIT(
    SEQ( COMMIT( Jim, ?default,
                ?default, doA ),
        COMMIT( Anna, ?default,
                ?default, doB ) ) )
```

This construct is discussed further in the context of commitment management in section 4.1.

- **ADOPT_GOAL** - The Goal Adoption Construct is a mental action which adds a goal to the agent's goal set, returning immediately. Let us consider a robot which has been requested to close the door. The agent can adopt the general goal of causing the door to be closed with the following commitment rule:

```
ADOPT_GOAL( GOAL( closed( door ) ) );
```

- **ACHIEVE_GOAL** - The Goal Achievement Construct adds the goal to the agent's mental state, but does not return until the GOAL is achieved.

3.1.10. Plans All of the plan operators above allow a plan to be constructed out of pre-defined actions. We refer to such constructions as *Plan Bodies*. To facilitate code reuse, plan bodies can be wrapped in plan constructs. The plan construct is similar to the actuator construct, but takes a plan body as activity, rather than a piece of external application code. The plan operators introduced above can operate on these plans as well as actuators, thus allowing the recursive definition of plans. From our tea making scenario, a (naive) plan to make a cup of tea can be represented as follows:

```
BEGIN{PLAN}
IDENTIFIER ie.ucd.assitant.makeTea();
PRE BELIEF(TRUE);
POST BELIEF(made_tea);
BODY SEQ(AND(getTea,getCup,getMilk),
        XOR(SEQ(getKettle,
                fillKettle,
                boilKettle),
        SEQ(getPot,
            fillPot,
            boilPot)),
        AND(addTea,pourWater),
        PAR(stir,addMilk),
        tasteTea);
END{PLAN}
```

3.2. Meta Constructs

In addition to the base language concepts discussed above, we have found that a number of extra constructs - that do not strictly form part of the underlying logical language - can vastly improve the usefulness of AF-APL. We now look at two of these *meta constructs*.

3.2.1. Modules An AF-APL module provides a simple way of declaring an external block of code (Java class or C object/library) which can be used to share functionality and memory between different actuators and perceptors. The construct takes one argument, the name of the module to be loaded by an agent. For example, a Java library which implements a hash-map, might be declared with the following:

```
MODULE myProject.mods.MyHashMap.class;
```

The module is instantiated and is agent specific, meaning all actuators and perceptors within an agent can have direct access to a common data source, which can be convenient when these actuators and perceptors need to share data, which is large in volume and not suited to being reasoned about by the agent.

3.2.2. Role Classes AF-APL provides a form of inheritance in agent design through the use of explicitly defined role classes. AF-APL Roles allow a collection of actuators,

perceptors and other agent components to be grouped together into an agent prototype. These agent prototypes can then either be instantiated directly into agents, or included in other agent designs. For example, the AgentFactory runtime [4] provides a number of pre-defined roles, including one that implements basic functionality for FIPA compliant communication. This role can be included into a new agent design with the USE_ROLE construct:

```
USE_ROLE ie.ucd.core.fipa.role.FIPARole;
```

3.2.3. Macro Inclusions To improve code reuse capabilities, Plans, Actuators, and Perceptors can all be coded directly in their own files, and later included into an agent with the AF-APL meta constructs USE_PLAN, USE_ACTUATOR, and USE_PERCEPTOR. For example, we can include the actuator for sending FIPA compliant messages as follows:

```
USE_ACTUATOR
    ie.ucd.af.fipa.fipaSend(?fipa_msg);
```

4. The AF-APL Execution Model

The AF-APL programming constructs, overviewed in the last section, must be related to each other and interpreted at runtime. We now discuss our execution model and commitment management system that is used to rationally drive an AF-APL agent. The agent's Life Cycle is built out of a number of Execution Cycles. AF-APL provides a well defined (but extensible) execution model which reflects the basic needs of a hybrid agent. Unlike traditional Sense - Plan - Act approaches, the model is asynchronous, guaranteeing the continued operation of an agent - regardless of potential locking of 3rd party actuator and perceptor code.

The AF-APL execution cycle is presented in pseudocode in figure 1. The first phase of the execution cycle concerns the update of the agent's belief set, \mathcal{B} , based on the results of perception and temporal belief update. To guarantee that the agent's execution cycle is not perturbed by potentially poorly designed 3rd party code, the perceptor firing and reading mechanisms are asynchronous. Actual perceptor code (e.g. C libraries) run outside of the agent's thread of execution; Instead, an interface to a perceptor is provided that includes a Belief Queue and an Event Queue. At this first phase of the execution cycle, each perceptor's Belief Queue is emptied into \mathcal{B} ; followed by the sending of a *triggered* event to advise the native perceptor code that it is time to re-fill its Belief Queue. Handling of the perceptors is then followed by temporal belief update. For each of the agent's temporal beliefs, the agent's beliefs are updated in accordance with the semantics of the temporal operators used.

The second phase of the execution cycle then follows (line 7), causing reactive actions to be run in their own threads, if the reactive rule condition is entailed by the \mathcal{B} .

```

1  FOR_EACH  $p \in \mathcal{P}$ {
2       $\mathcal{B} < p.queue;$ 
3      trigger(p);
4  }FOR_EACH  $b \in \mathcal{BR}$ {
5      IF b.condition {
6          add(b, $\mathcal{B}$ );}
7  }FOR_EACH  $r \in \mathcal{RR}$ {
8      IF r.condition {
9          trigger.action;}
10 }FOR_EACH  $g \in \mathcal{G}$ {
11     IF g.state == achieved {
12         drop(g);
13     }ELSE_IF g.state == new {
14         deliberate(g);
15     }ELSE_IF g.state == plan_found {
16         adopt(g.plan);}
17 }FOR_EACH  $cr \in \mathcal{CR}$ {
18     IF cr.condition {
19         add(cr.Commit, $\mathcal{C}$ );}
20 }FOR_EACH  $c \in \mathcal{C}$ {
21     process(c);
22     IF c.state == fulfilled ||
23         c.state == redundant {
24         drop(c);}
25 }
```

Figure 1. AF-APL Execution Cycle

The invocation of reactive rules is immediately followed by the deliberation phase. Our design of this phase was influenced by our wish to provide our agents with goal directed reasoning, while guaranteeing the agent's continued reactivity. These requirements immediately rule out direct means-end reasoning in the agents execution thread. Two alternative approaches were investigated. The first saw the execution of a number of means-end reasoning steps per execution cycle, where the semantics of means-end reasoning were directly defined in AF-APL. The second saw the use of an external planning algorithm which runs in an asynchronous manner similar to actuator execution. Of these two choices, practical considerations directed us towards the later option, since it allows more freedom in the design and implementation of the planning algorithm. During this phase of the agent execution cycle, each of the agent's goals are analysed. If a goal has been achieved, then it, along with any subsequent commitments, are immediately dropped from the \mathcal{G} and \mathcal{C} respectively. If a goal is new, then means-end reasoning is triggered asynchronously; and, if a solution has been found through means-end reasoning, then a commitment to the resultant plan is adopted. A drawback to the use of an asynchronous planning model is that by the time the planning process has returned, the plan may be invalidated by changes in the real world or agent state. We argue that this is a problem that will be common to any

agent acting in the real world, and is not specific to an asynchronous execution cycle.

Following deliberation, an AF-APL agent then addresses the questions of which commitments are to be adopted, and how these commitments are to be fulfilled. At a high level, this Commitment Management process involves the adoption of new primary commitments based on the values entailed by the agent’s mental state; followed by the partial achievement of these commitments through the management of commitment structures, and execution of actuators. Our approach to commitment management is detailed in the following section.

4.1. A Model of Commitments & Commitment Management

In section 3.1.4 we briefly introduced our notion of a commitment as a promise that is made by an agent, to perform an action, for some agent (possibly itself), More precisely, an AF-APL commitment, $c \in \mathcal{C}$, is a 6-tuple:

$$c = \{\alpha, \theta, \mu, \pi, \rho, \omega, \gamma\}$$

where α is the name of the agent to which the commitment has been made; θ is the earliest time at which the agent will attempt to fulfil the commitment; μ is the maintenance condition of the commitment; π is the activity that has been committed to; ω is an ordered set of child commitments; ρ is the state of the commitment; and γ is the commitment’s parent (potentially null). For brevity, α and θ can be viewed from a common sense interpretation. μ is a belief sentence - or conjunction of beliefs - which must be obeyed by the agent for as long as the commitment is to be held by the agent; If at any point this belief is not entailed by \mathcal{B} , then the commitment is dropped.

π is the activity which has been committed to by the agent. In our model, π is a placeholder for an actuator, plan, or plan operator; has its own state; and can generate a number of child activities. The semantics of an activity are directly dependent on the semantics for the plan operator, plan, or actuator object as appropriate. The states of an activity are: NEW, an activity has been initialised; ACTIVE, the actuator, plan, or plan operator is active; SUCCEEDED, the activity has succeeded; FAILED, the activity has failed.

The state of a commitment ρ signifies at what stage of fulfilment the commitment is at. The commitment’s state is important both from a conceptual view, in that it represents whether we are actively achieving a commitment, or if the commitment is planned to be achieved at a later time. It is also used in the commitment management model presented in the next section. A commitment can have five non-trivial states: DEPENDENT, a commitment that has been adopted,

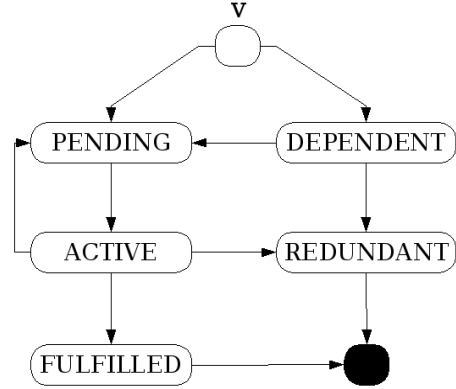


Figure 2. Commitment State Diagram

but is not due to be actively fulfilled until some fruition constraint has been achieved; PENDING, a commitment that the agent will begin to actively fulfil by the agent in its next execution cycle; ACTIVE, a commitment that is being actively pursued by an agent; REDUNDANT, a commitment that has failed - the maintenance condition for the commitment no longer holds; FULFILLED, a commitment that has been fulfilled - the activity committed to has succeeded.

Each commitment can have a number of child commitments’ ω , where each child is a commitment to some activity, which if achieved, can partially achieve some part of the parent commitment. For example, a commitment to perform two actions, can be refined to two commitments to more primitive actions. Thus, a commitment structure can be formed, where a coarse grained commitment can be broken down into sub-structures of finer grained commitments. We call the topmost commitment in a commitment structure a primary commitment. The agent’s commitment set \mathcal{C} contains a number of these commitment structures, where each structure was added through the initial adoption of a primary commitment through the application of commitment rules or deliberation.

4.1.1. Commitment Management Above, we gave a description of an AF-APL commitment and showed the relationship between primary commitments, activities, and commitment structures. In this section, we look at AF-APL’s commitment management algorithm. Once per execution cycle, each of the agent’s commitment structures are refined, which involves the pruning and expansion of the structure. A pseudo-code representation of our commitment management algorithm is presented in figure 3.

During the first refinement phase, the algorithm descends to the leaves of the commitment structure, and examines the state of each activity committed to. If a commitment’s activity has succeeded, then the commitment’s state is set to FULFILLED. Else, if the activity has failed, the com-

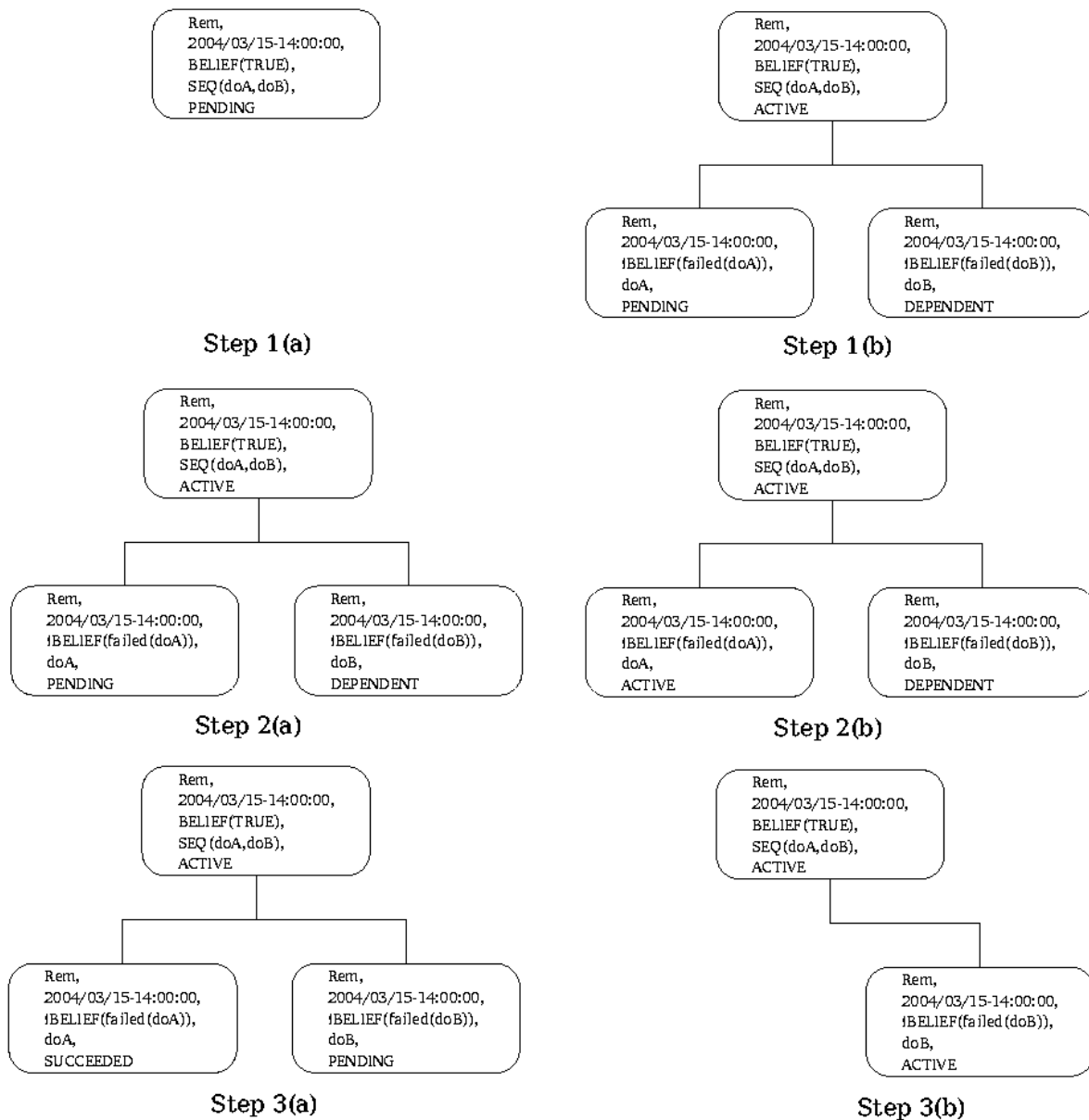


Figure 4. Commitment Structure Management

mitment's maintenance condition is checked. If this is no longer entailed by \mathcal{B} , then the commitment enters the REDUNDANT state; otherwise, the activity is reset. If the activity is still active, the maintenance condition is similarly checked - failure of the maintenance condition results in the commitment becoming REDUNDANT. By default, the leaf commitments, like all secondary commitments are created with a *weak minded* maintenance condition, meaning that if the committed to action fails, then the commitment enters the REDUNDANT state. The checking of mainte-

nence condition, and notification of parent commitment's activities percolate up through the structure until all commitments have been checked. As parent commitments are notified of the change of state of child commitments, other previously DEPENDENT child commitments can become moved to the PENDING state. Such transitions are dependent on the semantics of the activity committed to. It should be noted that since AF-APL uses an open-minded commitment model, a commitment fails if the commitment's maintenance condition no longer holds - not if the activity com-

```

1 process_commitment(c){
2   prune(c);
3   expand(c);
4 }
5
6 prune(c){
7   IF c.ω.size == 0{
8     IF c.π.state == SUCCEEDED{
9       c.ρ = FULFILLED; }
10    ELSE IF c.π.state == FAILED{
11      IF c.μ THEN c.ρ = PENDING;
12      ELSE c.ρ = REDUNDANT;
13    }ELSE{
14      IF !c.μ THEN c.ρ = REDUNDANT; } }
15  ELSE{
16    FOR_EACH child ∈ c.ω{
17      prune(child);
18      IF c.π.state == UPDATED{
19        c.ω.update(); }
20      ELSE IF c.π.state == SUCCEEDED{
21        c.ρ = FULFILLED; }
22      ELSE IF c.π.state == FAILED{
23        IF c.μ THEN c.ρ = PENDING;
24        ELSE c.ρ = REDUNDANT; } } }
25 }
26
27 expand(c){
28   IF c.ρ == ACTIVE{
29     FOR_EACH child ∈ c.ω{
30       expand(child); } }
31   ELSE IF c.ρ == PENDING{
32     IF c.θ {
33       Set c.ρ = ACTIVE;
34       Set c.π.state = ACTIVE;
35       FOR_EACH activity ∈ c.π.ω{
36         c_new = new c(activity);
37         add(c.ω, c_new); } } }
38 }

```

Figure 3. AF-APL Commitment Management Algorithm

mitted to fails. Thus, if a commitment’s activity fails, but the maintenance condition still holds, then the commitment is moved back to the PENDING state.

During the second refinement phase, all PENDING commitments are moved to the ACTIVE state, with a subsequent transition of the commitment’s activity from the NEW to the ACTIVE state. The consequence of moving the commitment’s activity to the ACTIVE state, is to create any new secondary commitments, marking these new commitments as DEPENDENT, ACTIVE, or PENDING as appropriate. The exact actions that are performed by during the setting of an activity to ACTIVE are dependent on

the semantics of AF-APL’s planning operators, plan and actuator constructs. A detailing of these semantics is considerably beyond the scope of this paper; but in summary, if the activity resolves to an actuator, then activation involves the triggering of the actuator in it’s own thread; If the activity corresponds to a plan identifier, then a child commitment to the plan body is adopted; If the activity corresponds to a plan operator (e.g. XOR, SEQ, TEST) then the creation of child commitments, is very much dependent on the semantics of the operator in question.

Although secondary commitment creation is, in general, dependent on the semantics of AF-APL’s activity types, the values for a secondary commitment’s agent α , earliest start time θ , and maintenance condition μ are specified by the basic commitment model; the default values for each secondary commitment’s α and θ are directly inherited from their parents, while the default μ for each agent is that the activity being committed to has not failed. These default secondary commitment creation values can be overwritten through the use of the CHILD_COMMIT construct introduced in Section 3.1.9. Figure 4 shows the first three commitment management steps following the triggering of the following commitment rule by Rem:

```

BELIEF(request(doStuff))
=> COMMIT(?Self,
          ?now,
          BELIEF(TRUE),
          SEQ(doA, doB));

```

In the first step, we see the initial adoption of the primary commitment to the SEQ(doA,doB), followed by the expansion of the primary commitment to a commitment structure. In the second step, doA is moved from the PENDING to the ACTIVE state - consequently triggering the external actuator code for doA (if we assume doA does in fact represent an actuator). In the third step, we assume that the doA actuator has successfully returned, thus causing the commitment to doA to be moved to the SUCCEEDED state, with the commitment to doB hence moved from the DEPENDENT to the PENDING state. If shown, this PENDING commitment would then be moved to ACTIVE in a subsequent commitment management step.

Our commitment management model is motivated by the dual constraints of allowing an agent to reason about its commitments, while guaranteeing the reactivity of the agent. For example, with our model, when a plan becomes active, then the commitment structure is fully expanded immediately, allowing an agent to reason directly about the commitments in the commitment structure.

5. Interpreting AF-APL: AgentFactory

In this section we introduce the AgentFactory Development Framework - along with its AF-APL compliant agent

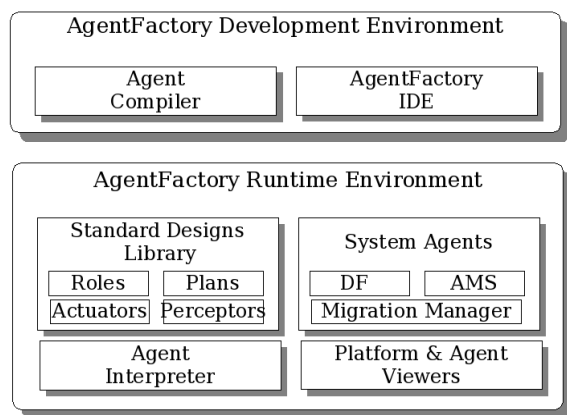


Figure 5. The AgentFactory Framework

interpreter¹. The focus of this paper is on the AF-APL language rather than any one interpreter design. Therefore, this overview is brief and intended to elaborate on how an AF-APL interpreter might be implemented.

The AgentFactory Framework [3, 4] is a complete agent prototyping environment for the fabrication of deliberative agents. As shown in figure 5, AgentFactory can be conceptually split into two components: the AgentFactory Runtime Environment (AF-RTE) and the AgentFactory Development Environment (AF-DE). The AF-RTE includes an agent interpreter; a library of standard actuators, perceptors, and plans; a platform management suite, which can hold a number of running agents at any time; and an optional graphical interface to view and manipulate agents and the platform. The AF-DE includes an agent/role compiler along with an Integrated Development Environment.

The *AgentFactory* interpreter uses factory instantiation to allow more than just AF-APL agents to be run. This was achieved by designing the interpreter around a number of interfaces - where each interface manages one of the most common component types in agent designs. Figure 6 gives an abstraction of the interpreter design. In our design, an AF-RTE platform can have any number of agents running - each agent is processed directly by an interpreter instance that is accessed through an Agent Core object. During interpreter initialisation, the Agent Core class uses a platform configuration file to instantiate the appropriate objects for a given agent type, thus allowing agents designed in different languages to be instantiated on the same platform. In addition to providing abstractions of mental attribute types, the interfaces (typefaced in italics) also abstract the agent controller. Thus, although the interpreter does not support

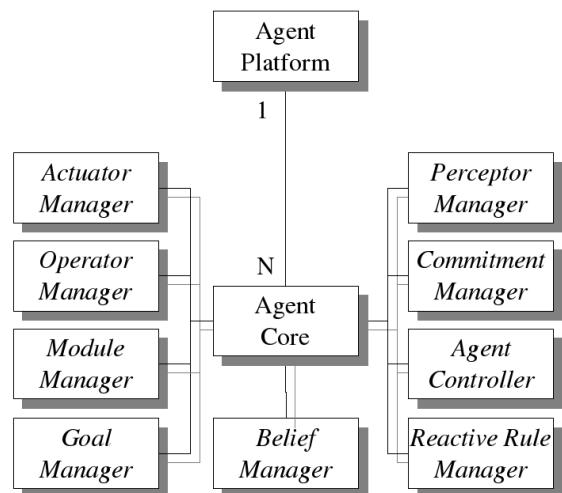


Figure 6. AgentFactory Interpreter Design

scripting of agent controller designs, new controller variants can easily be implemented and tested.

AF-APL, as described in this paper, is only one of a number of AO language variants that can potentially be instantiated on the platform. We have found that employing a flexible interpreter design is extremely beneficial for testing language variants, particularly when formal validation is not feasible.

6. Example Code

For illustration, we now present some code for a typical scenario that has influenced AF-APL's current features. Figure 7 presents a program fragment for an office assistant robotic agent. Starting from the top, we first see the use of the `USE_ROLE` macro to import AF-APL roles that provide domain specific definitions for FIPA compliant communication, and control of an autonomous wheelchair robot. The `USE_ACTUATOR` macro is then used to import a pre-defined actuator for vocalising utterances. This is followed by an in-line plan definition to deliver a parcel to a destination. The plan makes use of the `TRY_RECOVER` plan operator to initiate social help for recovery in the event that the basic delivery plan fails. The basic delivery plan also makes use of the `DO_BEHAVIOUR_UNTIL` operator which will cause the agent to follow the wall until it believes that a door has been found.

A commitment rule used states that if the agent believes that it has been asked to deliver a package, and if it believes that a superior made the request, and that the agent is not already committed to some other task, then the agent should immediately blindly commit to a simple plan. The plan uses an actuator inherited from the `FIPARole` to send

¹ The AgentFactory Framework is available for download, along with full documentation on it and the AF-APL language, from <http://www.agentfactory.com>

an acknowledgement message to the requesting agent, before using deliver plan, to perform the delivery task. Upon arrival at the destination, the agent will announce its presence. In addition to the commitment rule, a reactive rule is also specified; this simple rule states that if the agent believes there is an object blocking its active path, then it will initiate a reactive action to dodge the obstacle.

7. Six Lessons Learned

Here, we list a number of lessons learned from our practical application of AOP - particularly in the robotics domain - and show how AF-APL has been formulated to benefit from these lessons.

- **Provide an Extensible Core Language** An AO language should have a minimal conceptual footprint, while supporting extensibility based on domain specific requirements. AF-APL's actuator and perceptor interfaces act as membranes to domain specific code and abilities. This allows us to provide useful middleware features such as communication, yellow pages support, and migration, without the need to define these features within the semantics of the core language.
- **Provide a Practical Set of Planning Operators** Real agents often need to perform a number of tasks in complex arrangements that go beyond parallel and sequential constructions. AF-APL's new plan operator set provides an improved, but modest, set of constructs to facilitate the construction of complex plans. We acknowledge that any set of operators is going to be found incomplete in some way, but AF-APL has been designed such that we can easily extend the semantics of the language through the addition of new or updated planning operators.
- **Support Reasoning on Goals & Commitments** It is often extremely practical for an agent to reason about its own commitments, while deliberating over which new commitments should be adopted. AF-APL provides agents with these abilities through the well defined management of, and reasoning on commitment structures.
- **Support Robustness in Agent Operation** Agent programs must guarantee the continued reactive behavior of an agent - despite the possibility of bugs, crashes, or hanging in 3rd party actuator or perceptor code. All aspects of AF-APL's execution cycle are asynchronous, insuring localised degradation in the face of 3rd party code.
- **Provide Multiple Levels of Reasoning** Agent programming should provide a spectrum of control levels,

ranging from full goal oriented deliberation through to reflective action. In addition to providing traditional Goal and Commitment directed deliberation, AF-APL allows for reflective behavior within individual actuator or perceptor implementations; these are in addition to Reactive Rules that provide reactive abilities within the intentional layer.

- **Support Code Re-Use** For AOP to become useful in the real world, common software engineering practices will need to be supported by the languages. AF-APL provides code re-use facilities through the use of Role Inheritance and macro definitions and inclusions.

8. Conclusions & Future Work

We presented AF-APL as a language based on a formal agent treatment - but forged through lessons learned in application experiences. A key feature of the language is its asynchronous execution model that guarantees robust operation of the agent, even in the face of potentially poorly designed 3rd party code. Also, the language utilises a model of commitments that allow an agent to reason about its own future actions and goals in the pursuit of rational action.

AF-APL was originally given a formalisation around Possible Worlds Semantics. However, with the practically derived features presented here, we are currently in the process of re-formalising, based around an Operational Semantics. From a practical standpoint, we must further investigate the uses of roles and inheritance within AF-APL. We hope that such an effort will result in a greater understanding of how rapid prototyping, inheritance, and code-reuse can help to proliferate the AO paradigm.

Acknowledgements - We gratefully acknowledge the support of the Deutschen Forschungsgemeinschaft (DFG) through the SFB/TR 6023 project on Spatial Cognition - Subproject I3-SharC.

References

- [1] M. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, USA, 1987.
- [2] P. Cohen and H. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.
- [3] R. W. Collier. *Agent Factory: A Framework for the Engineering of Agent Oriented Applications*. PhD thesis, University College Dublin, 2001.
- [4] R. W. Collier, G. O'Hare, T. Lowen, and C. Rooney. Beyond prototyping in the factory of the agents. In *3rd Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'03)*, Prague, Czech Republic, 2003.
- [5] M. Dastani, F. Dignum, and J. Meyer. 3apl: A programming language for cognitive agents. *ercim news, european research*, 2000. Consortium for Informatics and Mathematics, Special issue on Cognitive Systems, No. 53,.

```

1 USE_ROLE ie.ucd.core.fipa.role.FIPARole;
2 USE_ROLE de.tzi.RollandRole;
3 USE_ACTUATOR de.tzi.MARY.speak(?utterance);
4
5 // Explicit Plan Definition
6 BEGINPLAN
7 IDENTIFIER de.tzi.safeDeliver();
8 PRE BELIEF(TRUE);
9 POST BELIEF(delivered(parcel));
10 BODY TRY_RECOVER(SEQ(DO_BEHAVIOUR_UNTIL(followWall,
11                                     BELIEF(found(Door))),
12                               enterDoor),
13                               social_recovery),
14 ENDPLAN
15
16 // Commitment Rule
17 BELIEF(requested(?agent,deliver(?object,?destination)))
18 & BELIEF(isSuperior(?agent)) & !COMMIT(?a,?b,?c,?d)
19   => COMMIT(?agent,
20             ?Now,
21             BELIEF(TRUE),
22             SEQ(fipaSend(?agent,inform(ack(deliver(?object,?dest))))),
23             safeDeliver,
24             speak("I have a parcel for collection")
25             );
26
27
28 // Reactive Rule
29 BELIEF(blocked(ahead)) & BELIEF(moving(forward)) => EXEC(dodgeObstacle);

```

Figure 7. Sample AF-APL Program

- [6] M. Dastani, B. van Riemsdijk, F. Dignum, and J.-J. Meyer. A programming language for cognitive agents: Goal directed 3apl. In *Proceedings of AAMAS 03*, 2003.
- [7] D. C. Dennett. *The intentional stance*. The MIT Press, Massachusetts, 1987. 388 pages, 1987.
- [8] I. Dickinson and M. Wooldridge. Towards practical reasoning agents for the semantic web. In *2nd Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS-03)*, Melbourne, Australia, July 2003.
- [9] M. Georgeff and A. Lansky. Reactive reasoning & planning. In *Proceedings of the Sixth International Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, WA, USA, 1987.
- [10] K. V. Hindrikis, F. de Boer, W. van der Hoek, and J. J. Meyer. Agent programming in 3apl. In *In Proceedings of Autonomous Agents & Multiagent Systems (AAMAS) 1998*, 1998.
- [11] D. Phelan, R. Strahan, R. Collier, C. Muldoon, and G. O’Hare. Sos: Accomodation on the fly with access. In *Proceedings of the 13th International FLorida Artificial Intelligence Research Symposium Conference (FLAIRS 2004)*, Miami Beech, Florida, 2004.
- [12] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: Implementing a bdi-infrastructure for jade agents. in: *EXP - In Search of Innovation (Special Issue on JADE)*, 3(2):76–85, September 2003.
- [13] A. Rao. Agentspeak(l): Bdi agents speak out in a logical computable language. In *proceedings of the Seventh European Workshop on Modelling autonomous agents in a MultiAgent world*, Institute for Perception Research, Eindhoven, The Netherlands, 1996.
- [14] A. S. Rao and M. P. Georgeff. BDI agents: from theory to practice. In V. Lesser, editor, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS’95)*, pages 312–319, San Francisco, CA, USA, 1995. The MIT Press: Cambridge, MA, USA.
- [15] R. Ross, R. Collier, and G. O’Hare. Demonstrating social error recovery with agentfactory. In *Proceedings of The Third International Joint Conference on Autonomous Agents and Multi Agent Systems*, 2004.
- [16] R. J. Ross, R. O’Donoghue, and G. O’Hare. Improving speech recognition accuracy on a mobile robot platform using top-down visual cues. In *Proc. 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, 2003.
- [17] Y. Shoham. Agent oriented programming. *Artificial Intelligence*, 60:51–92, 1993.