Running head: MODELING AND PROGRAMMING WITH COMMITMENT RULES

Modeling and Programming with Commitment Rules in Agent Factory

R.W. Collier

G.M.P. O'Hare

CLARITY Centre for Science, Engineering and Technology

School of Computer Science and Informatics

University College Dublin, Ireland

ABSTRACT

Please insert your abstract here.

Modeling and Programming with Commitment Rules in Agent Factory

Introduction

Agent-Oriented Programming (AOP) is a relatively new programming paradigm introduced by Yoav Shoham (1993) in which software systems are viewed as consisting of a set of *agents* that interact with one another to solve problems that are beyond their individual capabilities. More specifically, agents are viewed as high-level autonomous software entities that encapsulate a set of capabilities and whose internal state is comprised of a set of mental components such as beliefs, capabilities, choices and commitments. This view of agents as mentalistic entities is a common perspective within multi-agent systems research and underpins many of the most prominent agent theories (Cohen & Levesque, 1990; Rao & Georgeff, 1991; Wooldridge, 2000). These theories model the internal decision-making process of an agent in terms of the interplay between the constituent components of the underlying mental state. Their objective is to define how an agent is able to act in a rational goal-directed manner and to tease out various desirable properties that emerge from that action. Thus, the objective of AOP is to present a framework for developing a new class of programming languages that are derived from these theories.

This view of programs that consist of components whose state is defined by a set of mental qualities and in which computation is realized through speech act based message passing is not unique to Shoham. John McCarthy, who is widely acknowledged as being one of the first to associate mental qualities with machines (McCarthy, 1979) had, as early as 1990, written an draft proposal for a programming language he entitled *Elephant 2000* (McCarthy, 1992). That said, McCarthy recognized at the time that his language could be implemented, and to this day, no implementation of Elephant 2000 exists (McCarthy, 2007). In contrast, a number of AOP languages have been proposed, implemented and successfully used to build a range of agent-oriented applications.

This chapter focuses on one such AOP language, entitled the *Agent Factory Agent Programming Language (AFAPL)* (Collier, 2001; Ross & Collier & O'Hare, 2004; Collier & Ross & O'Hare, 2005). AFAPL has its origins as part of a larger software engineering framework entitled *Agent Factory* (O'Hare, 1996; Collier, 1996; O'Hare & Collier & Conlon & Abbas, 1998; Collier, 2001; Collier & O'Hare & Lowen & Rooney, 2003). Both AFAPL and the associated framework have been designed specifically to support the fabrication of agent-oriented applications, and have been used extensively in the development of a number of prototype systems in areas such as: robotics (O'Hare & Duffy & Collier & Rooney & O'Donoghue, 1999; Dragone & Holz & O'Hare, 2007), mobile computing (O'Hare & O'Grady, 2003; Muldoon & O'Hare & Phelan & Strahan & Collier, 2003), virtual and mixed reality (Duffy & O'Hare & Campbell & Stafford & O'Grady, 2005), wireless sensor networks (Marsh & Tynan & O'Kane & O'Hare, 2004; O'Hare & O'Grady & Marsh & Ruzzelli & Tynan, 2006) and information retrieval (Peng & Collier & Mur & Lillis & Toolan & Dunnion, 2004; Lillis & Collier & Toolan & Dunnion, 2007).

The remainder of this chapter starts with an overview of Agent-Oriented Programming. This is followed by a general overview of the Agent Factory framework. The next section presents

details of AFAPL, and after that some of the tools that have been developed to support its use are discussed. The penultimate section, discusses future trends in this area, and finally, some concluding remarks are presented.

Agent-Oriented Programming

In his seminal paper, Shoham (1993) introduces the concept of Agent-Oriented Programming as apecialisation of Object-Oriented Programming (OOP). As is highlighted in Figure 1, the core computational unit becomes an agent rather than an object, and the state is constrained to mental components. The second significant restriction that is applied to AOP is the requirement that the only valid types of message are *speech acts*. Speech act theory (Searle, 1969) is a pragmatic theory of how humans communicate with one another, and has become the defacto standard within the multi-agent systems research community (Wooldridge & Jennings, 1995). Ultimately, the adoption of speech act theory has led to the specification of various *Agent Communication Languages (ACLs),* such as Knowledge Query Meta Language (KQML) (Finin & Labrou & Mayfield, 1997) and Foundation for Intelligent Physical Agents (FIPA) ACL (FIPA, 2000) which form the basis for the communication infrastructure of most modern agent systems.

Based on this view of AOP, computation arises through a combination of message-passing and mental state update. Practically, this is achieved through the implementation of an *agent interpreter* that updates the agents' mental state and manages the passing of messages in accordance with some associated *agent program*. Specifics relating to the operation of this interpreter and the structure of the associated agent program are bound intimately to the particular agent theory that is chosen to underpin the corresponding programming language.

The first AOP language to be developed, entitled *Agent-0*, was intended as illustrative example (Shoham, 1993), and is described in more detail later. Agent-0 was followed closely by the *PLAnning Communication Agents (PLACA*) language (Thomas, 1995), which attempted to address the lack of support for plans and planning within Agent-0. Specifically, PLACA introduced capabilities as first class constructs within the language, and modeled the intentions of the agent rather than its commitments. Execution of the internal state thus involves matching

	OOP	**AOP**
Basic Unit	Object	Agent
Parameters describing state of basic unit	Unconstrained	Beliefs, capabilities, choices, commitments, …
Process of computation	Message passing and response methods	Message passing and response methods
Types of message	Unconstrained	Inform, request, offer, promise, decline, …
Constraints on methods	None	Honesty, consistency, …

Figure 1: Table showing OOP versus AOP [Shoham, 1993]

capabilities to intentions through the formation of plans of action via a classical STRIPS (Fikes & Nilsson, 1971) style planning process. In contrast with Agent-0, which was made available for use, no prototype of PLACA was made public.

Around the same time, Arnand Rao proposed the *AgentSpeak(L)* language (Rao, 1996), which built on his extensive theoretical and practical work in the area of Belief-Desire-Intention (BDI) agents (Rao & Georgeff, 1995). Unfortunately, Rao's language was only a proposal and was not implemented until around 2004 when *Jason*, a Java based development environment for AgentSpeak(L) was created (Bordini & Hubner & Vieira, 2005). However, during this interim period, Hindriks & de Boer & van der Hoek & Meyer, (1999) developed *3APL*, a rule-based programming language that is inspired by the BDI-architecture and which has many similarities with AgentSpeak(L). 3APL is the first AOP language whose syntax and semantics was formally specified using operational semantics (Plotkin, 1981), and while it is similar to AgentSpeak(L), Hindriks et al. (1999) point out that 3APL offers a richer set of rules than AgentSpeak(L) as it not only provides rules for matching goals to plans, but also provides support for modeling the modification and revision of goals. The 3APL interpreter has been implemented in Java and has been made available for use.

Agent-0

In this section, we take an in-depth look at the first AOP language, Agent-0. Our motivation for doing this is two fold: firstly, Agent-0 is the first AOP language to be developed, and secondly, its design is centered around the notion of commitment to action as opposed to the more goal-driven BDI approach has become more common. Given that AFAPL also adopts a more commitment centric approach, it would seem natural and appropriate to focus on Agent-0.

As is common in multi-agent systems research, Shoham formalizes the operation of Agent-0 through the definition defines a set of modalities that he uses to formally define the agent interpreter. These modalities act upon a first-order logic that is augmented with a time component. In this logic, Shoham augments both the base facts, and the modalities with times, for example: `finished(rem, chapter)`^{t} represents that rem finished this chapter at time t; and `BEL`$^{t}_{a}$ `finished(rem, chapter)`$^{t'}$ represents that at time t, agent a believes that rem finished this chapter at time t'. Shoham justifies this representation of time by stating that: "we believe things both about different times and at different times, and the same is true of other modalities" (Shoham, 1993).

Shoham models actions through as being defined implicitly, so that, "rather than say that the robot took the action `raise-arm` at time t, we will say that the sentence `raise-arm(robot)`^{t} is true" (Shoham, 1993). When conjoined with modalities this allows an agent to believe it is capable of raising its arm or to believe that it has raised its arm, or even allows the agent to believe such things about other agents.

Given this model of time and action, Shoham defines three fundamental modalities: *belief*, *obligation*, and *capability*. The first modality, which was introduced informally above, enables an agent to have beliefs about what it thinks the current state of the environment it, and potentially beliefs about those and other agents' beliefs. The second encompasses the notion of commitment to action, in that it represents the commitment of one agent to another at a given time to believe a certain state of affairs. Finally, the third modality defines states of affairs an agent can bring about at a given time. For example, an agent can have the capability at time 5 to

open the door by time 8.

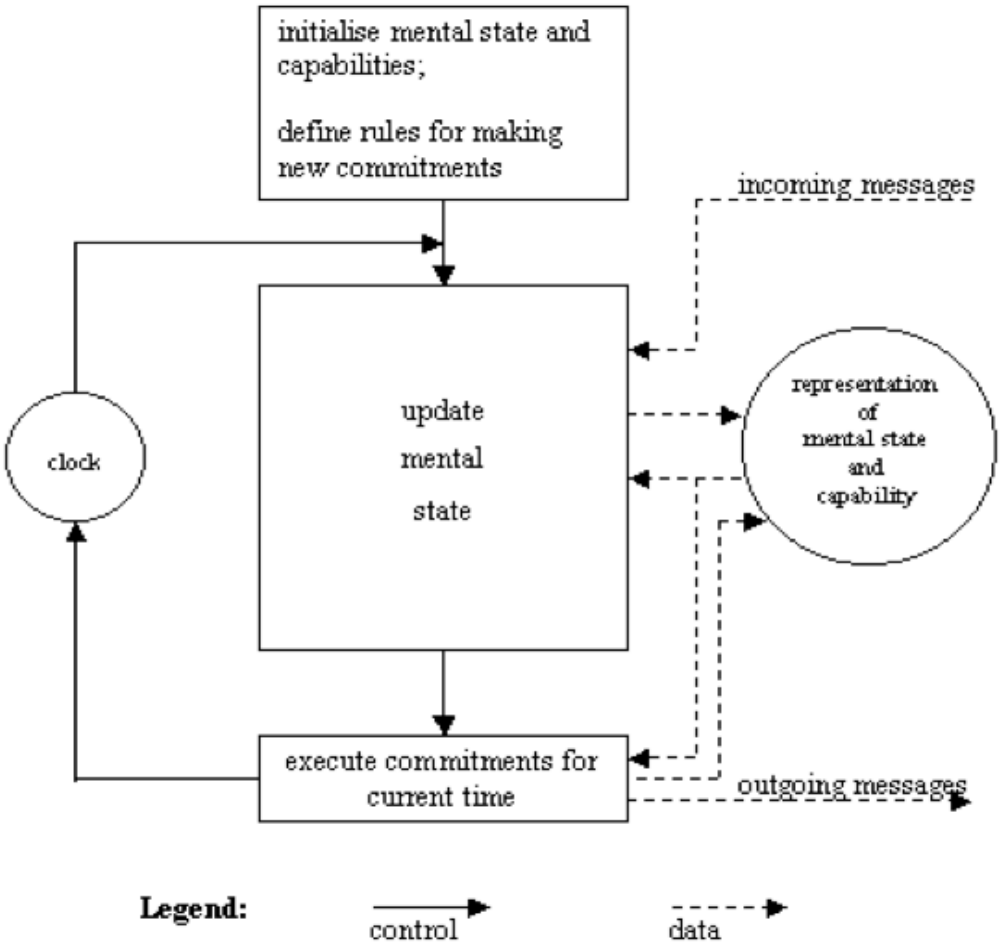

Figure 2: Schematic of the Agent-0 Interpreter (taken from [Shoham, 1993])

Two additional modalities, *decision* and *able*, are defined as extensions of obligation and capability respectively. Decision represents obligation to oneself, and able represents an "immediate" version of capability. By immediate Shoham means those capabilities that can be carried out at that time point, for example if an agent is able to open the door at time 5, then at time 5 it has the capability to open the door immediately.

Defining the above set of modalities provides a formal model of the agents' mental state. However, they provide little insight into the nature of the agent interpreter. This is provided through the ascription of various properties to this model, such as, *internal consistency*, *introspection*, and *persistence of mental state*.

Shoham implements this model through an agent interpreter. His design is illustrated in figure 2. The properties of the model are encoded within the *update mental state* phase of the interpreter cycle, and the representation of the mental state and capability encodes the modalities (in Agent-0, belief is represented by the B keyword, and obligation by the CMT keyword – representing commitment).

The behaviour of Agent-0 agents is programmed via a high-level programming language. This language defines syntax for the representation of positive and negative facts based upon standard LISP semantics, for example `(NOT (t (employee jones acme)))` represents the fact that *jones is not an employee of acme at time t*. Actions may be distinguished by type, *private* or *communicative*, and activation, *conditional* or *unconditional*. Thus actions types are distinguished by the keywords DO (for private), and INFORM, UNINFORM, REQUEST, UNREQUEST, or REFRAIN (for communicative). Unconditional actions take the form `(DO t p-action)` which states that at time t, the agent should perform the private action p-action. Conditional actions are triggered through some mental condition. Mental conditions may be either a belief statement or a commitment (obligation) statement. Conditions in Agent-0 may include variable bindings.

Finally, the top layer of the Agent-0 language is a collection of *commitment rules* that define when and how agents should adopt new commitments. These combine some mental condition with some message condition and define a corresponding action that should be committed to should the conditions arise. As a result, it is these rules that define the overall behaviour of the agent and, while these behaviours are not defined in terms of goals that should be achieved, they do support the definition of undesirable situations in which the agent should act.

As Shoham acknowledges, Agent-0, was never intended to be a fully-fledged AOP language, but rather a prototype that could be used as a basis for other researchers. As a result, Shoham leveled a number of criticisms at both Agent-0 and AOP. We conclude this section be outlining some of the more main criticisms that are directed at AOP languages.

- *Extensions to the Paradigm.* The relation between AOP and OOP should be further investigated to consider additional properties of OOP such as inheritance, composition, and polymorphism. Informally, this would also implicitly require the introduction of some form of reuse mechanism.
- *Extensive Tool Support.* It is vital that developers wishing to use an AOP language have access to the same quality and variety of tools that they would have access to if developing an OOP program. This includes integration with established IDE's, such as Netbeans and Eclipse, and the development of equivalent tool sets for AOP languages,
- *Clear Link to the Environment.* Any practical AOP language must specify clearly how agents interact with their environment. Lack of clarity can result in confusion for developers, which will impact on the potential uptake of AOP languages.
- *Integration with Current Best Practices for Agent Systems.* Many of the earlier AOP languages were either not distributed, or made use of some form of bespoke infrastructure. Since the emergence of the FIPA Standards (FIPA, 2000) many of the agent toolkits have become, at least to some level, compliant. AOP languages must offer similar levels of compliance in order that any agents developed will be able to interoperate with other agents developed using other toolkits.

Armed with these general criticisms of AOP languages, the next three sections introduce the Agent Factory framework, a constituent AOP language, known as AFAPL, and some of the tools that have been developed to support the coding of agents using AFAPL.

The Agent Factory Framework

Agent Factory (AF) is an open source framework that has been developed to support the deployment of multi-agent systems (Collier et al., 2003). It is broadly compliant with the current FIPA standards and is implemented in the Java programming language. In its latest incarnation, the framework has been restructured to facilitate the deployment of applications that consist of multiple agents that employ a diverse range of agent architectures. In this the framework has become an enabling middleware layer that can easily be extended and adapted for different application domains.

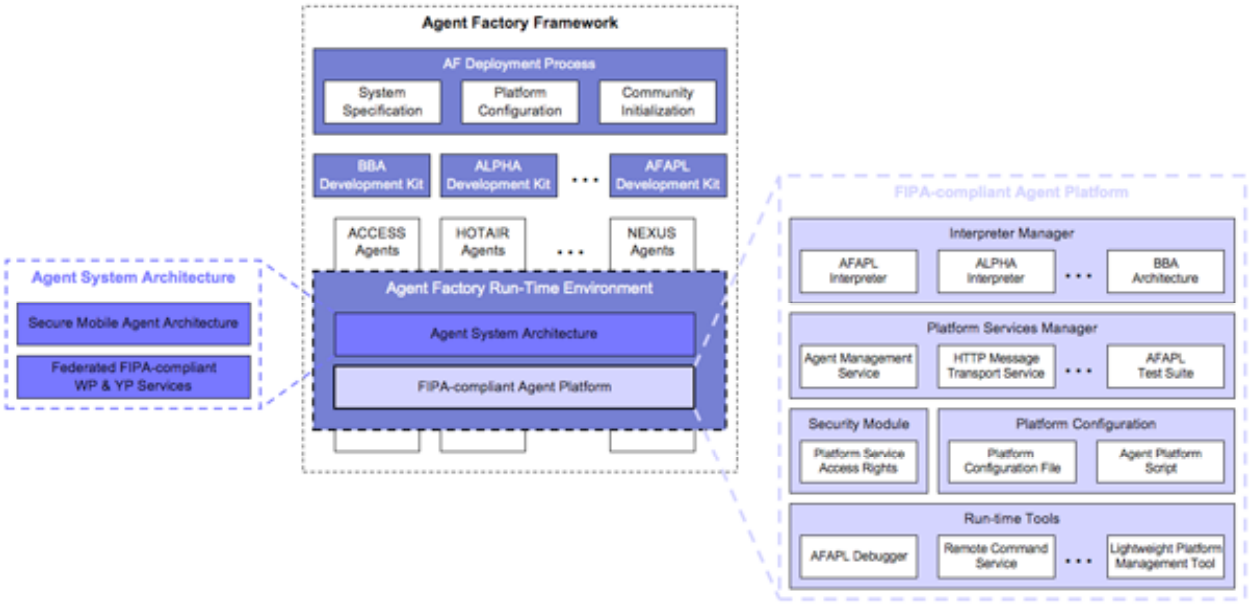

Figure 3: A Schematic of the Agent Factory framework

As can be seen in the schematic of AF presented in Figure 3, the overall framework has been broken down into three parts:

- a distributed *Run-Time Environment (RTE)* that consists of a FIPA-compliant *agent platform* together with a number of prefabricated *agent system architectures* (software architectures);
- a set of *Development Kits (DK)* that contain agent interpreter / architecture implementations together with relevant tool support; and
- a *Deployment Process* that provides a structured approach to the deployment of multi-agent systems using the framework.

Of these, the two core components are the agent platforms and the DKs. Specifically, each agent platform contains the machinery necessary to create and support agents that are developed using one or more of the DKs. This machinery includes:

- an *agent container* that holds the agents currently resident on the agent platform;

- a manager that supports the deployment of a set of *platform services* that implement shared resources that are available to some or all of the resident agents;
- a *security module* that controls what platform services each agent has access to;
- a module that manages the set of *agent interpreters / architectures* for the various types of agent that are currently deployable on the platform; and
- a module that creates and deploys any *run-time tools* that are required to provide necessary support for the visualization, management, profiling, and debugging of the resident agents.

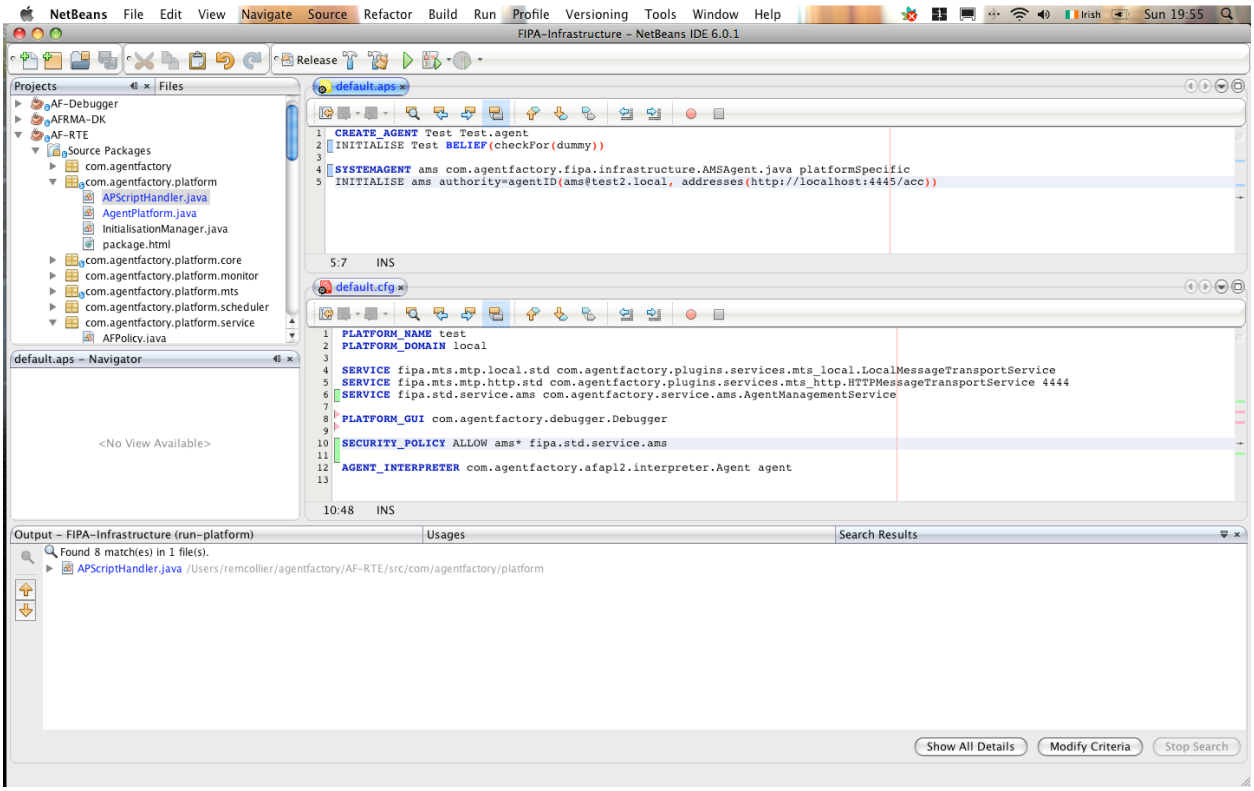

Figure 4: Screenshot of the Netbeans IDE Plugin for Agent Factory. The upper pane shows an open platform configuration file, while the lower pane shows an open agent platform script file.

The configuration of the agent platform and the specification of the initial agent community are supported via two deployment files: the *platform configuration file* (with a .cfg extension) is used to specify a configuration for each agent platform and the *agent platform script* (with a .aps extension) is used to specify the initial community of agents that are to be deployed on that platform. Examples of these two file types are given in the screenshot in figure 4, which showcases a plugin for the Netbeans Integrated Development Environment. This plugin is also integrated with some of the tool support that comes as part of the AFAPL-DK (the development kit for AFAPL). This is described in a later section.

As can be seen in the example presented in Figure 4, each agent platform is given a unique name and is associated with a domain. In this case, the agent platform name is “test.local”. Platform services are associated with a platform via the SERVICE command. As part of their declaration, each service is given a unique identifier (in the case of the HTTPMessageTransportService and

AgentManagementService, this is specified in the associated FIPA specifications) together with the name of the class that implements the service. For configuration purposes, optional settings can also be specified after the class name. For example, the HTTPMessageTransportService in the above example is configured to work on port 4444.

Platform services are shared resources that are, by default, accessible by every agent that is currently residing on the platform, although agents must explicitly bind to a service in order to access it. One kind of platform service that is an exception to this is a Message Transport Services (MTS). These services are used to support inter-agent communication, and deal with low-level issues of sending and receiving messages. All agents are automatically bound to all available MTS platform services when they are created. Currently, four MTS are provided as part of the framework: the *Local-MTS* provides intra-platform agent communication via Java message passing; the *HTTP-MTS* provides a HTTP-based message transport service that is based on the corresponding FIPA specification; the *UDP-MTS* provides a UDP-based message transport service that works on a multicast channel, but does not guarantee delivery; and the *MPP-MTS* provides a SMTP-style client-server based message transport service that can be used to circumvent firewalls.

The restriction of access to individual platform services is supported through the use of security policies. For example, in the above platform configuration file, a security policy is defined that restricts access to the AgentManagementService, a second FIPA service that provides support for the run-time management of agents (i.e. creation, suspension, resumption, and termination), so that only agents whose name beings with “ams” can access the service. Once a security policy is applied to a platform service, the default policy for the service becomes denial of access to that service.

Finally, the platform configuration file also allows multiple agent interpreters (or architectures) and run-time tools to be associated with an agent platform via the AGENT_INTERPRETER and PLATFORM_GUI keywords respectively. As with the SERVICE keyword, optional configuration information may be appended to the end of the mandatory set of parameters. This information is passed directly to the underlying Java implementation for the associated platform service, agent interpreter or run-time tool. Support for the development of these components is provided in the form of Java APIs, which include abstract base classes that may be extended to implement the new feature. Details of how to do this are provided via various development guides that are available on the Agent Factory website.

The agent platform script file is used to declare the initial agent community that will be deployed on an agent platform at startup. This file allows developers to define what agents will be created, to initialize the state of those agents, and where to start their execution. The script allows the creation of two kinds of agents: *system agents* and *application agents*. System agents are those agents that make up the system architecture, while application agents are those agents that contain the application logic. In the example presented in Figure 4, one system agent is created, with a platform specific name that begins with ams (platform specific names are generated by appending the specified name with an @ symbol followed by the full platform name). One application agent is created with the name Test. The main practical differences between system agents and application agents are that system agents are created first and have the chance to carry

out initial configuration before the application agents are created, and system agents start executing by default, while application agents must be started explicitly via the START_AGENT command. Initial configuration information may be passed to either kind of agent via the INITIALISE keyword and the format of the data passed is specific to the underlying agent architecture.

The mapping between agent program and agent interpreter / architecture is based on the file extension used in when the agent is created. For example, in the above example, the Test agent is created using a file with a ".agent" extension which is used to identify its implementation as AFAPL target code). In cases where the architecture is implemented directly in Java, as is the case with the ams agent, the ".java" extension is used.

In summary, Agent Factory provides a modular and extensible framework for constructing and deploying multi-agent systems. It offers a set of APIs that can be used to implement and integrate agents that employ different architectures and can support multiple interpreters that realise high-level AOP languages via the creation of Development Kits. One such language, which is described in the next section, is the *Agent Factory Agent Programming Language.*

The Agent Factory Agent Programming Language (AFAPL)

Agent Factory (AF) provides support for the programming of agents through AFAPL, a purpose-built AOP language that is inspired by Agent-0. AFAPL builds on previous work presented in (O'Hare, 1996) and (Collier, 1996) in which VDM (Bjorner & Jones, 1978) was used to specify an agent interpreter for AF. Further, the specification presented in (O'Hare, 1996) was subsequently refined into an algebraic specification that was implemented using the UMIST Paradox System (Powell, 1989), while the specification presented in (Collier, 1996) was implemented in the Smalltalk-80 programming language. Underlying the latter implementation was a purpose-built configuration language that supported the definition of agents that utilized the associated interpreter. Following on from these initial implementations, this configuration language has evolved into AFAPL. As part of this evolution, (Collier, 2001) formally defines the syntax and semantics using a multi-modal branching time first-order logic of commitment. The language itself combines the mental attitudes, whose syntax and semantics are defined via the logic, with additional programming constructs that support the specification of the interface between the agent and its environment. This has resulted in a practical AOP language that is implemented in Java, and has been successfully applied in a number of significant problem domains.

In contrast with many of the more recent AOP languages, AFAPL does not adopt the Belief-Desire-Intention (BDI) mental state architecture as its core. Instead, it employs a commitment-based mental state that is similar to the one used in the design of Agent-0. However, in contrast with Agent-0, AFAPL offers the following:

- Support for the definition of composite activities based on plan operators together with a mechanism for declaring, at design time, a set of partial plans (via a *plan library*).

- The implementation of a well-defined model of commitment management that specifies how commitments are adopted, refined, maintained and realised; and what to do in the event that a commitment fails.
- A set of tools, presented later, that provide comprehensive support for the development process.
- The provision of a well-defined interface between the agent and its environment that can be customised through the creation of *perceptors* (sensors) and *actuators* (effectors) and their subsequent declaration within the relevant AFAPL program.
- Support for the decomposition of agent programs into reusable units together with an import mechanism that is similar to #include in C.

In its current incarnation, AFAPL is a logic-based language in which an agent is viewed as a computational entity whose core mental state consists of *beliefs, plans* and *commitments*. Beliefs represent the agent's (possibly incorrect) knowledge of the state of the environment; plans are complex activities that may be performed by the agent, and which are combinations of plan operators and other (more primitive) activities; and commitments represent the activities that the agent has decided to perform. These three mental attitudes are joined by a set of *commitment rules*, which encode the situations in which the agent should adopt new commitments. In the following sections, we discuss these core concepts, the corresponding programming constructs and their associated managers in more detail. The syntax and semantics of AFAPL have been formalized via a multi-modal first-order branching-time logic (Collier, 2001). In this section, we present a detailed, but informal, overview of both the language and the operation of the associated aspects of the agent interpreter.

Beliefs and Belief Management

The agent's beliefs represent the (possibly incorrect) knowledge that an agent has about itself and its environment. Knowledge takes the form of facts that are represented as well-formed formulae of an underlying domain language that is based on first-order logic. For example, the fact that Rem has finished ironing can be represented as: `finished(Rem, ironing)`. As is usual for such logics, variables may be introduced to specify unknowns, for example, the fact that someone has finished ironing can be represented as: `finished(?x, ironing)`. Building on this, the beliefs of an agent are represented as a set of grounded belief literals, which are themselves well-formed formulae of the underlying domain language. As is typical for such languages, these well-formed formulae are "wrapped" within an operator that classifies them as beliefs. For example, the belief that Rem has finished ironing is represented as: `BELIEF(finished(Rem, ironing))`.

A key decision made in the design of the belief model employed in AFAPL was to make beliefs transitory. That is, beliefs do not persist beyond the iteration of the interpreter cycle in which they are adopted. The motivation for this arises as a result of the use of *perceptors* as the core belief update mechanism. Perceptors are part of the environment interface of an agent and they are used to convert sensor data gathered from the environment into beliefs. For example, in the robot soccer domain, a perceptor may be used to generate a belief about the presence or absence of a ball within the robot's field of vision. This belief may be used to drive the overall strategy of the robot and result in a change in its underlying behaviour such as switching from a "defensive"

behaviour to a “ball tracking” behaviour. Similarly, other agents may need to be aware of the current time, and as a result, may adopt beliefs about the current time. Rather than attempt to specify which kinds of beliefs should persist or to require that perceptors also manage the removal of existing beliefs; the approach adopted in AFAPL is to remove all beliefs at the beginning of the iteration and to build a new belief set via activation of the agents associated set of perceptors. This approach is inspired by the foundational approach to belief revision in that the perceptors generate a set of foundational beliefs at the start of each interpreter cycle. These foundational beliefs, which are termed the *current beliefs* of the agent, are then augmented through a set of *belief rules* that define a set of derived beliefs that may be inferred from the foundational beliefs.

As is highlighted in the upper part of Figure 5, perceptors are instances of concrete Java classes that extend the `com.agentfactory.core.agent.Perceptor` abstract base class, and implement the perceive() method. This abstract base class facilitates access to the associated agent implementation together with a number of helper methods that simplify access to shared resources (i.e. platform services), and simplify the adoption of beliefs. The example perceptor in Figure 5 generates beliefs about the agents name and the current iteration. When implemented, perceptors must be explicitly associated with an agent program through the PERCEPTOR construct. An example of a perceptor declaration is presented in the lower part of Figure 5.

```
// The Java implementation
public class SelfPerceptor extends Perceptor {
    public void perceive() {
        adoptBelief("BELIEF(name(" + agent.getName() + "))");
        adoptBelief("BELIEF(iteration(" agent.getIteration() + "))");
    }
}

// The AFAPL declaration
PERCEPTOR self {
    CLASS SelfPerceptor;
}
```

Figure 5: An example Perceptor and the corresponding AFAPL declaration

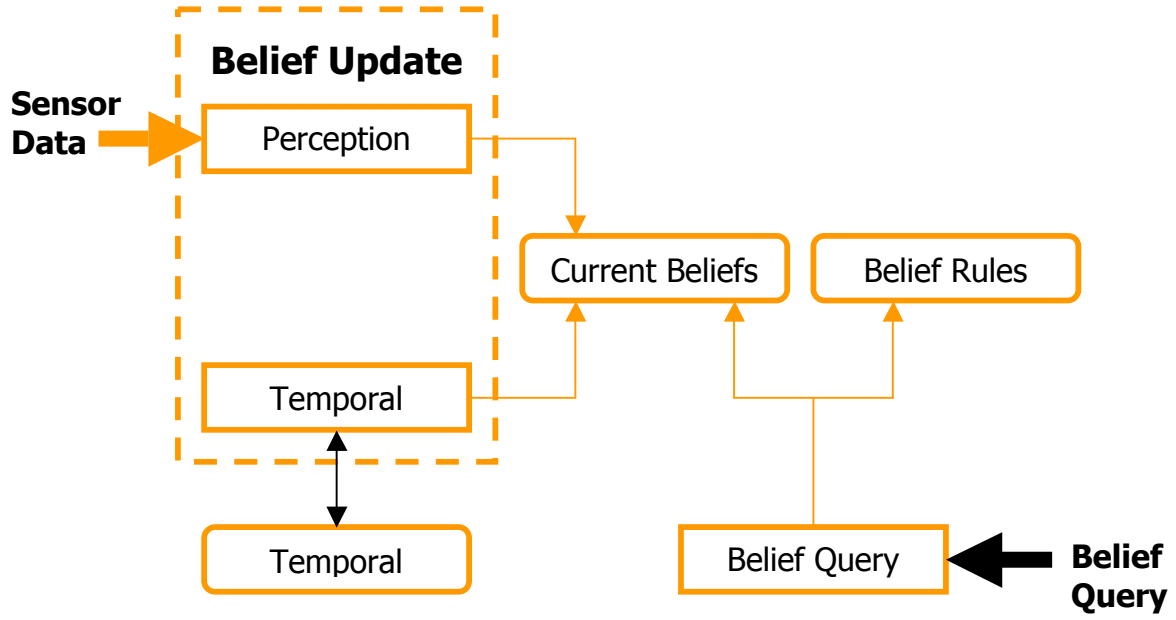

Figure 6: Belief Management in AFAPL

By adopting this model, the current beliefs of the agent are directly generated based on sensor data that is gathered from the agent's environment, the belief set that the agent uses to reason about how to act is, as far as possible, accurate with respect to that environment. However, the downside is that the beliefs of an agent become akin to a short-term memory, and information cannot be easily maintained over multiple interpreter cycles. In response to this, the AFAPL belief model includes a set of *temporal beliefs* that can be used to represent more persistent beliefs. Along with current beliefs and belief rules, this third component of the belief model holds beliefs that are associated with one of three temporal operators:

- the NEXT operator is used to define beliefs that will should be adopted at the start of the next interpreter cycle, for example, `NEXT(BELIEF(finished(Rem, ironing)))` represents the belief that Rem will have finished the ironing by the next iteration;
- the ALWAYS operator is used to define beliefs that persist for the lifetime of the temporal belief, for example, `ALWAYS(BELIEF(overworked(Rem)))` represent the belief that Rem is always overworked; and
- the UNTIL operator is used to define beliefs that persist until some associated condition arises, for example, the temporal belief `UNTIL(BELIEF(doing(Rem,ironing)), BELIEF(finished(?someone,ironing)))` represents the belief that Rem is doing the ironing, and this belief is persists until the agent believes that someone (here represented by the variable) has finished ironing.

Temporal beliefs are kept separate from the current beliefs of the agent. The effect of adopting a temporal belief is that the associated belief is also adopted as a current belief. For example, if the agent adopts the temporal belief, `ALWAYS(BELIEF(overworked(Rem)))` then the agent also adopts a current belief, `BELIEF(overworked(Rem))` on each iteration. As is illustrated in Figure 6, the adoption of current beliefs based on the temporal beliefs of the agent is managed

```
ONTOLOGY health {
    TERM timeToCheck(?x, ?y);
    TERM healthy(?x);
    TERM failed(?x);
}
```

Figure 7: An example AFAPL Ontology

via a temporal update step that is part of the belief update process. This step is preceded by a perception step in which the agent fires its perceptors. The perceptions generated by these perceptors are added to a *belief update set*, which also includes any beliefs adopted by actions that were performed during the previous interpreter cycle.

The current beliefs of the agent are used as a basis for making decisions about how the agent should act. This is supported through the concept of a *situation*. Informally, a situation is a conjunction of positive and negative belief literals that should be evaluated with respect to the current beliefs of the agent. For example, we can represent the situation that somebody is hungry and there is some food to eat as: `BELIEF(hungry(?x)) & BELIEF(hasFood(?y)).` Situations are evaluated through a belief query process (see Figure 6), which provides a resolution-based (Robinson, 1965) query mechanism through which the truth or falsity of the situation can be evaluated. The implementation of the belief query component is based on the reasoning mechanisms employed in Prolog interpreters, and is applied to both the current beliefs and the belief rules of the agent. In addition to assessing the truth or falsity of a situation, the belief query process can also return a set of all the variable bindings that, if applied, would cause the situation to be evaluated to true.

Finally, AFAPL includes additional support for defining valid terms in the underlying content language. This is done through the use of *ontologies*. Ontologies are specified using the ONTOLOGY construct and can be used to define a set of terms for the agent program. For example, Figure 7 presents a simple ontology for a Health Monitoring agent that includes thee terms:

- `timeToCheck(?x, ?y)` is a term that is used to represent the fact that it is time to perform a health check on the agent with name ?x and addresses ?y;
- `healthy(?x)` is a term that is used to represent the fact that ?x is healthy; and
- `failed(?x)` is a term that is used to represent the fact the ?x has failed.

Initially, support for ontologies was introduced as a mechanism for detecting semantic errors during the compilation of agent programs. As such, a minimalist approach was adopted. It is anticipated that future versions of AFAPL will expand on this model to offer integration with more established ontology languages such as OWL.

Actions and Plans

The actions and plans of an agent describe the set of activities that the agent is capable of performing. Actions refer to the primitive capabilities of the agent that are directly executable via an associated *actuator* unit, which forms the second part of the interface between the agent and its environment. In contrast, plans refer to the complex capabilities of the agent and consist of a set plan operators that define orderings on an underlying set of activities, which may themselves also be either actions or plans.

Actuators implement the primitive capabilities of an agent. They are instances of Java classes that extend the abstract `com.agentfactory.core.agent.Actuator` base class, and implement the act(…) method. Again, this base class provides access to the agent implementation together with a number of helper methods that simplify access to shared resources, and the adoption of beliefs. Specifically, as was indicated in an earlier section, beliefs that are adopted via actuators are added to a *belief update set*, which forms the basis of the agents current beliefs for the next iteration.

To illustrate how actuators work, the upper part of Figure 8 presents an example actuator that can be used to dynamically create another agent at run-time. As can be seen, this actuator retrieves a reference to the AgentManagementService platform service, and invokes the createAgent(…) method on this service, taking the name and type of the agent to be created from the parameters of the action identifier, which is passed as a parameter to the act(…) method. The method then implements the action, returning a Boolean value to indicate whether or not it succeeded.

```
// The Java implementation
public class CreateAgent extends Actuator {
    public void act(FOS action) {
        String name = action.argAt(0).toString();
        String type = action.argAt(1).toString();

        AgentManagementService service = (AgentManagementService)
              getService("fipa.std.service.ams");
        service.createAgent(name, type);
        Service.resumeAgent(name);
        adoptBelief("BELIEF(created(" + name+ "))");
        return true;
    }
}

// The AFAPL declaration
ACTION createAgent(?name, ?type) {
    PRECONDITION BELIEF(true);
    POSTCONDITION BELIEF(created(?name, ?type));

    CLASS CreateAgent;
}
```

Figure 8: An example Perceptor and the corresponding AFAPL declaration

Similarly to perceptors, actuators must also be explicitly declared within the relevant AFAPL agent program. As is shown in Figure 8, this is achieved though the use of an ACTION statement. This statement defines an action in terms of an identifier (that is used to refer to the action in the context of commitments), a pre-condition that must be satisfied before the action may be executed, a post-condition that will be satisfied after the action is completed, and the class that implements that action. Informally, when the agent realizes an activity to an action, it has a unique identifier for that activity that is based on the associated action identifier, for example `createAgent(Bob, mydesign.agent)`, refers to an instance of the `createAgent` action. When the underlying actuator is invoked, this identifier is passed to the act(…) method, where the data stored within the arguments can be extracted.

In addition to action, AFAPL also includes support for the declaration of plans. A plan is basically a composite activity that combines a set of sub-activities whose order of realization depends upon an associated plan operator. Sub-activities may themselves by either actions or plans. Currently, AFAPL supports the following plan operators:

- **SEQ(act_1, …, act_n)**: Defines a sequence of activities that should be performed. When realising this plan, the agent performs each activity consecutively, so act_1 is performed before act_2 and so on.
- **PAR(act_1, …, act_n)**: Defines a set of parallel activities that can be performed at the same time. When realising this plan, the agent tries to perform each activity at the same time. This may not happen in cases where the preconditions associated with a given activity are not met.
- **OR(act_1, …, act_n)**: Define a set of activities, only one of which must be completed successfully for the plan to be considered successful. For example, if the agent were to perform the plan **OR(A, B)**, then the agent need only complete either A or B. Once a sub-activity is successful, all other attempts to complete the plan become redundant.
- **DO_WHEN(cond, act):** Defines a conditional activity that is only carried out when (if) the corresponding condition is satisfied. This plan operator can be effectively combined with the OR plan operator to implement a choice mechanism (i.e either do A when X or B when Y).

```
PLAN healthCheck(?name, ?addr) {
    PRECONDITION BELIEF(true);
    POSTCONDITION BELIEF(true);

    BODY
      SEQ(request(agentID(?name, ?addr), ping),
        OR(DO_WHEN(BELIEF(fipaMessage(inform, sender(?name, ?ad), pong)),
             adoptBelief(BELIEF(healthy(?name)))),
           SEQ(DELAY(milliseconds(5000)),
             adoptBelief(BELIEF(failed(?name))))));
}
```

Figure 9: An example AFAPL plan for performing a health check

- **DELAY(time):** Defines a delay of a prescribed time period. The specified time may be either a time in milliseconds or a number of iterations.
- **ATTEMPT(X, Y, Z)**: Defines an plan where the agent should attempt X and, if X is successful, should do Y, otherwise the agent should do Z.
- **FOREACH(cond, act):** Defines a plan where the agent does the action for each variable binding that satisfies the condition.

While explicitly declared plans are sufficient for defining the behaviour of agents, however, there are number of cases where the same plan appears in a number of places. In order to facilitate the reuse of such plans, AFAPL supports the definition of partial plans via the PLAN construct. This construct is the counterpart (for plans) of the ACTION construct in that it associates each partial plan with a unique identifier and a set of pre- and post- conditions. Informally, the pre-conditions define the situation that must be satisfied for the plan to be realized, and the post conditions define the expected outcome of following the plan successfully.

Finally, the identifier can include a number of parameters that can be used to customise the underlying plan. For example, Figure 9 shows a plan for carrying out a health check on another agent whose name is "?name" and who is contactable by a set of addresses denoted by "?addr". The plan works by sending a "ping" request message to the specified agent that is being monitored, and waiting for a "pong" inform response message. If the agent received the pong message before a deadline of five seconds (5000 milliseconds), then the agent adopts a belief that the monitored agent is healthy otherwise it adopts a belief the monitored agent has failed.

Both actions and plans are realized through the adoption of commitments to the relevant action or plan and the subsequent realization of that commitment. Details of this are presented next.

Commitments and Commitment Management

Commitments represent the outcome of the agent's decision-making process. Specifically, a commitment is defined to be the mental equivalent of a contract, which describes what the agent has decided to do, for whom it has decided to do it, when the chosen activity must be realized, and the terms and conditions under which the commitment may be dropped (known as the *maintenance condition*). For example, consider a Health Monitoring agent that has been charged with the task of monitoring the health of a number of application agents by "pinging" those agents on a regular basis via the use of the `healthCheck(…)` plan that was described in Section 4.2. At a certain point in time, the agent will decide to perform the health check on one of the agents that is is monitoring. In the case where this is the agent known as AnAgent, then the agent would adopt a commitment, shown in the upper part of Figure 10, to the `healthCheck(…)` plan.

The agent adopts this commitment through the firing of a special type of rule, known as a *commitment rule*. Commitment rules define the situations in which the agent should act. These situations are evaluated with respect to the current beliefs of the agent during each iteration of the interpreter cycule. Should the situation arise, then the agent will adopt a commitment the corresponding commitment. By way of illustration, the lower part of Figure 10 includes an example commitment rule, which states that, if the agent believes that it is time to check the health of a certain agent (whose name and contact addresses are known), then the agent should

adopt a commitment to the `healthCheck(…)` plan. In addition to the variable defined in the situation, AFAPL also includes two additional system variables that may be used in a commitment rule: ?self is bound to the name of the agent, and ?now is bound to the current time. To illustrate how commitment rules work, consider the case where the agent believes: `BELIEF(timeToCheck(AnAgent,addresses(http://localhost:4444/acc)))`. Based on this belief, the commitment rule would be triggered, and the agent would adopt the commitment in the top part of Figure 10 (assuming the agent was called HealthMonitor and the current time was 11:32).

Our use of commitments is in contrast with the more popular BDI architecture (Rao & Georgeff, 1995) in which the future activities of an agent are driven by its desires, which represent the ideal state of the environment, and intentions, which are a chosen subset of those desires, which the agent considers to be achievable and which it is actively trying to achieve. An agent achieves its intentions by following a chosen course of action. How the agent selects that course of action is an implementation issue.

```
// Example Commitment
COMMIT(HealthMonitor, 11:32, BELIEF(true),
    healthCheck(AnAgent, addresses(http://localhost:4444/acc)));

// Example Commitment Rule
BELIEF(timeToCheck(?name, ?addr)) =>
COMMIT(?self, ?now, BELIEF(true), healthCheck(?name, ?addr));
```

Figure 10: An example of a Commitment and a Commitment Rule in AFAPL

Typically, BDI-based AOP languages, such as AgentSpeak(L) and 3APL, adopt an approach in which intentions are represented by the plans that the agent decides to perform. Our use of commitments is somewhat similar. That is, commitments represent the courses of action that the agent has decided to try and achieve. The key difference between commitments and intentions is that commitments represent a decision to do something, they do not necessitate that the agent intends to bring about the resulting state of affairs, nor do they necessitate that the agent has a corresponding goal to bring about some future state of the world. The agent may simply be following a social convention that it must respond to requests from other agents, or may be reacting to some change in the environment.

Further, by adopting the concept of a commitment as the primary mechanism for representing the choices of the agent, we are able to associate a strength of commitment with that choice. The idea of commitment strength originates with the notion of a *commitment strategy* (Rao & Georgeff, 1991). In their work, a commitment strategy is a property of the underlying decision-making aparatus of an agent, and it defines how committed the agent is to its intentions. To illustrate this, they identified three potential commitment strategies that an agent could employ:

- *Blind Commitment:* The agent is committed to its intentions until it achieves them.
- *Single-Minded Commitment:* The agent is committed to its intentions only so long as it believes them to be achievable.

- *Open-Minded Commitment:* The agent is committed to its intentions only so long as they still congruent with its goals.

In AFAPL, rather than forcing all commitments to adhere to one of the above strategies, we allow the developer to specify how committed the agent is to a given commitment through the inclusion of a *maintenance condition*. In cases where the agent should be blindly committed to some course of action, this mainteance condition is set to true. On the other hand, if the maintenance condition is set to define when the commitment is no longer achievable or (implicily) is no longer a goal of the agent, then the agent can become either single-minded or open-minded about that commitment.

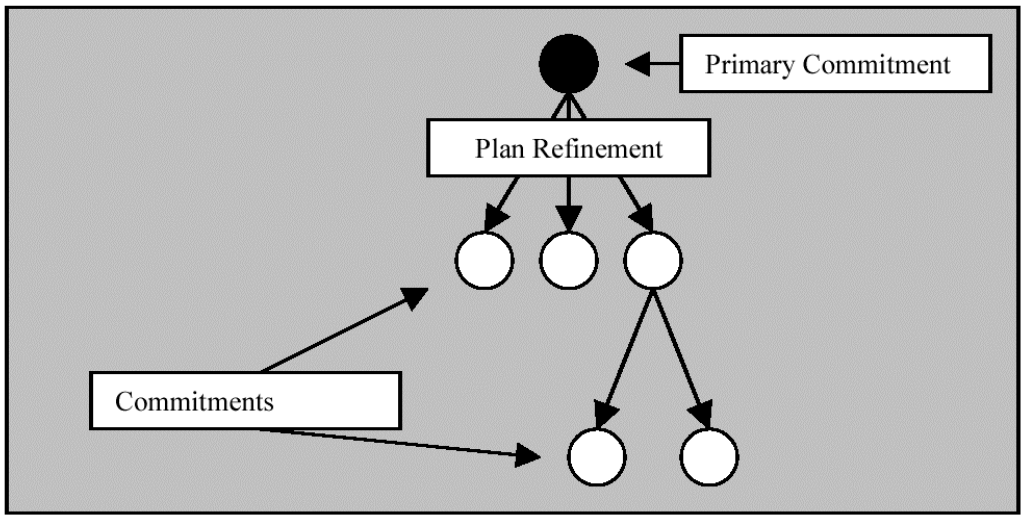

Figure 11: A View of the Hierarchical Relationships within a Commitment Structure

The notion of a commmitment strategy, when applied directly to commitments, can be extended further to consider how more than just how a commitment is maintained. In general, such an expanded strategy can be termed a *commitment management strategy* (Collier, 2001). A commitment management strategy specifies a set of sub-strategies that define how an agent: (1) adopts new commitments, (2) maintains its existing commitments, (3) refines commitments to plans into additional commitments, (4) realizes commitments to primitive actions, and (5) handles failed commitments. These strategies realised through a meta-level process, known as *commitment management*.

The principal sub-strategy that underpins the behaviour of AF-APL agents is *commitment adoption*. Commitments are adopted either as a result of a decision to realise some activity, or through the refinement of an existing activity. The former type of commitment is known as a *primary commitment* and the latter as a *secondary* (or *tactical*) *commitment*. Primary commitments are adopted as a result of the triggering of a commitment rule,while secondary commitments, on the other hand, are adopted in response to the agents' efforts at realizing an existing commitment to a plan of action. That is, secondary commitments are adopted as part of the *commitment refinement* process that is enacted whenever an agent attempts to realize an activity to a plan. The secondary commitments correspond to each of the sub-activities that the plan operator has been applied to.

The commitment refinement process forces a hierarchical relationship between commitments that can be modelled as a tree-like structure as is presented in Figure 11. However, this structure is complicated by the additional constraints that are placed on secondary commitments as a consequence of the semantics of plan operators. For example, if an agent has committed itself to a plan that is a sequence of activities, then, when it refines this commitment into a set of secondary commitments, it can only consider the secondary commitment that corresponds to the first activity in the plan. That is, the sequential plan operator creates a dependency that limits when the agent may try to realize the secondary commitments. Similar dependencies are created for the other plan operators and are discussed in (Collier, 2001) along with further details of how failure of commitments are handled by the AFAPL interpreter.

Interpreter Cycle

Agent Factory includes a built-in scheduler that organizes the execution of all of the agents residing on a given agent platform. Agents are allotted a time slice by the scheduler, and the interpreter cycle is started when that time slice begins. As can be seen in Figure 12, the interpreter cycle consists of five basic steps: belief update (UPDATE), commitment adoption (ADOPT), commitment management (MANAGE), action (ACT), and migration (MIGRATE). The interpreter is designed so that the first three steps are guaranteed to complete no matter how small the time slice is. The action step then uses as much of the remainder of the time slice as possible / necessary, sequentially executing actions that the agent has decided to perform. The final migration step simply checks whether certain conditions have arisen, and if so, contacts the migration service so that migration can start. We describe each of these steps in more detail below, mapping them onto relevant sections of the chapter.

As was discussed earlier, the belief update process employed in AFAPL has been decomposed into two sub-processes: perception (PERCEPT) and temporal update (TUPDATE). The first of these steps refers to the firing of the agent's preceptor units and the generation of a up-to-date snapshot of the current state of the environment. Conversely, the second step augments this snapshot through the evaluation of a set of persistent beliefs.

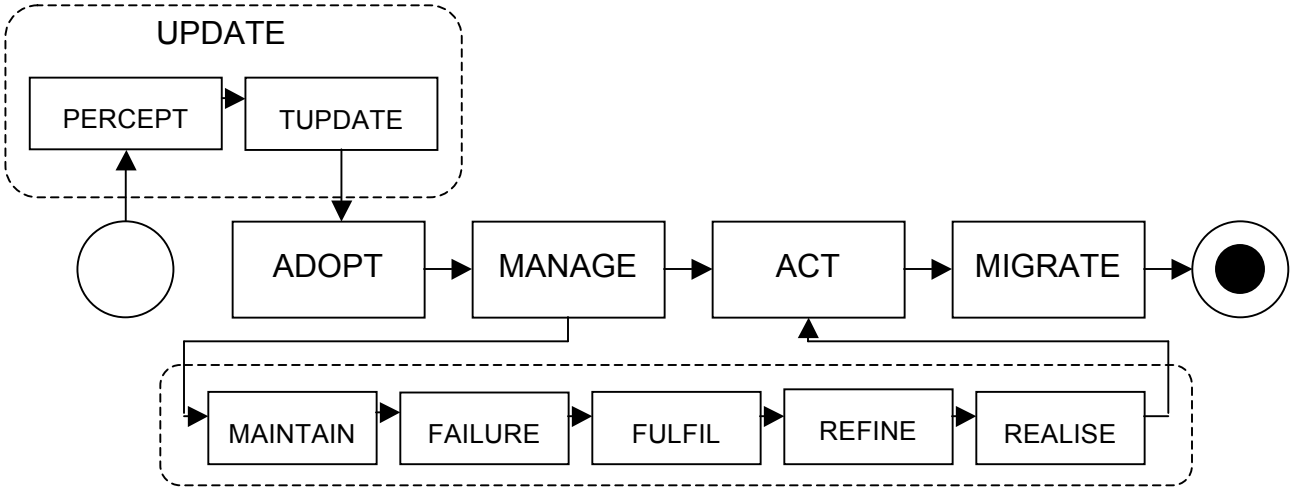

Figure 12: Overview of phases in the AFAPL Interpreter

The commitment adoption (ADOPT) is in reality part of the commitment management (MANAGE) step. However, as was explained earlier, it is this part of the commitment management strategy that governs the overall behaviour of the agent as it is through the adoption of commitments, via the firing of commitment rules, that the agent decides what activities to perform. Consequently, it has been given first class status in our interpreter algorithm. In contrast, the commitment management (MANAGE) step of the algorithm can, if required, be further decomposed the following five phases:

- *Commitment Maintenance* (MAINTAIN): The maintenance condition of each commitment is checked, and those commitments whose condition is no longer true with respect to the beliefs of the agent are considered to have failed.
- *Commitment Failure Handling* (FAILURE): The failure of a commitment is propagated to its parents and children (if they exist). While all child commitments are set to a failed state, and the impact of a child's failure on the state of the parent commitment depends upon the activity associated with that commitment. For example, the failure of a child whose parent is a sequence (SEQ) plan results in the failure of the parent, while the failure of a child whose parent is a optional (OR) plan only causes the parent to fail if it is the last option and all previous options have also failed.
- *Commitment Fulfilment* (FULFIL): The commitments are checked, and any leaf commitments (those commitments that have no children) that have either succeeded of failed are dropped.
- *Commitment Refinement* (REFINE): Any commitments to plans of action are refined by adoption a one or more sub-commitments based on the plan operator that is being refined.
- A commitment to a plan template *pt* (identified by the plan identifier) is refined to a commitment to the plan specified in that plan template.
- *Commitment Realisation* (REALISE): Commitment realization applies only to commitments that are made to executing plan templates (specified by the plan identifier) or actions. Commitments made to these activities are realized when the time component of the commitment is satisfied and the preconditions of the corresponding activity are valid. For actions, the resulting behaviour is the scheduling of the commitment for execution, while, for plan templates it is the adoption of the underlying plan of action.

The action (ACT) step of the interpreter cycle deals with the actual execution of the agent's actions. During the commitment management phase, decisions are made as to which of the agents primitive activities should be realized. These activities are added to an internal schedule whose content is processed in this step. If enough time remains in the agents time slice, the action step removes items sequentially from the schedule in the order in which they were added and processes them one by one. This continues until either the schedule becomes empty or the agent's time slice finishes.

Finally, the migration (MIGRATE) step performs a check to see if the agent has decided to move to another agent platform. If this is the case, then the agent also checks to see if there are any commitments. If not, then the agent is removed from the scheduler and the migration platform service is notified, telling it to start the migration of the agent. In essence, migration in AFAPL starts by the agent performing the migrate(…) action, which sets an internal migration flag and stores details of the target platform that the agent will migrate to. Once the migration flag is set,

then the interpreter suppresses the commitment adoption step so that no new commitments are adopted. Only when the outstanding (existing) commitments are completed (i.e. the commitment set becomes empty) is the agent is ready to migrate.

Reuse of AFAPL Code

Support for the reuse of AFAPL code is provided in the form of an IMPORT statement that is based on the C #include statement. When building AFAPL programs, source code is written in a file that has a ".afapl" extension. These files are incorporated into the Java package structure, and can be referenced using standard Java naming conventions. For example, Figure 13 shows two example AFAPL programs where the second program reuses the code of the first program.

The first program builds on the health monitor example presented earlier, and implements the other half of the "ping" protocol (i.e. it defines how a monitored agent should respond to a health check). The second program is the AFAPL equivalent of the "Hello World" program. It defines a single action, called `alive`, whose implementation (actuator) will print out "Hello World!" to the console, and a single initial commitment to the `alive` action. The key point demonstrated here is the use of the IMPORT statement, which integrates the code for interacting with a HealthMonitor agent with the Alive agent program.

As is discussed later, the AFAPL compiler converts source code into a more machine-oriented target code, denoted by files with a ".agent" extension. As part of this process, the compiler combines all of the partial agent programs into a single block of code, which it then converts into the target code.

```
// Program: Monitored.afapl (in agent package)
BELIEF(fipaMessage(request, sender(?name, ?addr), ping) =>
COMMIT(?self, ?now, BELIEF(true),
    inform(agentID(?name, ?addr), pong));

// Program 2: Alive.afapl
IMPORT agent.Monitored;

ACTION alive {
    PRECONDITION BELIEF(true);
    POSTCONDITION BELIEF(true);

    CLASS actuator.Alive;
}

COMMIT(?self, ?now, BELIEF(true), alive);
```

Figure 13: Two Example AFAPL Programs: Monitored.afapl and Alive.afapl

The ability to write reusable blocks of program code is an essential requirement of any practical programming language. While the reuse mechanism presented here is quite simplistic, it has had a significant impact on the usability of AFAPL as it facilitates the creation of code libraries that can easily be reused. This has led to the creation of a number of basic AFAPL programs that act

as a starting point for developers. In this light, we have created the `com.agentfactory.core.fipa.agent.FIPACore` agent program, which contains the basic set of preceptors and actions required for agent communication. We have then built on this to create the `com.agentfactory.core.agent.BasicAgent` agent program which extends the core communication behaviours to include a number of core perceptors, actions and behaviours that we believe are core to all agents. This second program includes: actions for the adoption and retraction of beliefs; actions that support the binding to and unbinding from platform services; perceptors that generate beliefs about the current state of the agent platform; preceptors that generate beliefs about the current state of the agent (e.g. name, iteration); and so on.

AFAPL Tool Support

Support for the fabrication of agents using AFAPL is provided through a number of tools that are packaged as part of the AFAPL Development Kit (AFAPL-DK). These tools range from editors, compilers, and code libraries through to debuggers, test suites, and visual development tools. In the following sections, we highlight some of the more prominent tools in more detail.

Two tools not discussed in detail here are the plugin for the Netbeans IDE and the VIsual Protocol EditoR (VIPER). A screenshot of the Netbeans plugin was presented earlier in Figure 4. In addition to the support provided for deploying agents using Agent Factory, this plugin also includes support for creating and compiling AFAPL programs. This includes templates for all components, syntax highlighting for the AFAPL files, and integration with the AFAPL Compiler.

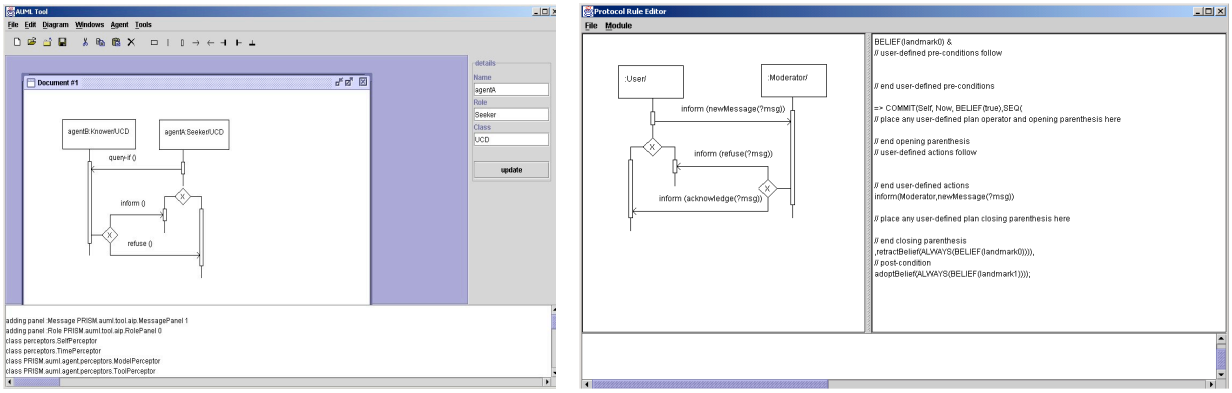

Figure 14: Screenshots of the VIPER Protocol Editor (Left) and Rule Editor (Right)

VIPER (Rooney & Collier & O'Hare, 2004) is a visual development toolkit that is based on Agent UML (Bauer & Muller & Odell, 1999) Protocol Diagrams. It is comprised of two tools: a Protocol Editor that provides a visual tool for generating Agent UML Protocol Diagrams, and a Rule Editor that further supports the user by guiding them through the step of implementing the protocols in AF-APL. Two screenshots depiciting both the Protocol Editor and the Rule Editor are presented in Figure 14.

The AFAPL Compiler

The primary responsibility of the compiler is to convert source code, denoted by a ".afapl" extension, into target code, denoted by a ".agent" extension that may be deployed. To achieve this, the compiler takes as input a source code file, and generates an AFAPL parse tree. Once the initial version of this tree has been generated, it is analysed, and any node relating to an IMPORT statement is expanded so that the resultant parse tree contains all of the source code associated with the target agent program. During this process, the detection of any syntactic errors results in the compilation process failing and an appropriate set of errors being displayed to the developer.

Following on from this, the compiler enters the first of two phases that, if completed successfully, ensure that the parse tree represents a syntactically, and as far as possible semantically, correct AFAPL program. In the first of these phases, it tries to collect various pieces of information about the program. Specifically, this phase of the compilation process involves two passes that:

- *Pass 1: Generate Ontology Database*. This pass searches the parse tree for nodes that contain ONTOLOGY constructs, and builds a temporary database of content language terms as defined in those ontologies.
- *Pass 2: Generate Activity Database*. This pass searches the parse tree for activity (i.e. action and plan) declarations and constructs a temporary database that contains the identifiers of each of the activities found.

In particular, the second pass extracts the identifiers of all activities (actions and plans) that are declared in the parse tree. A requirement of AFAPL is that each activity have a unique identifier. This requirement is enforced during this pass - if the compiler locates an activity identifier that is already in the activity database then the compiler halts and an appropriate error message is generated and displayed.

Once the compiler has collected this data, a second phase starts in which it carries out a series of checks that attempt to minimise the potential for semantic errors. Specifically, it performs two additional passes:

- *Pass 3: Check Beliefs*. This pass searches the parse tree for nodes that contain beliefs, and compares them against the terms that are stored within the ontology database.
- *Pass 4: Check Activities*. This pass searches the parse tree for plans and commitments and extracts all activity identifers from them. It then compares the extracted identifiers against the database of activities that was compiled in pass 2.

Once these fours passes have been successfully completed, the compiler generates the resultant target code for the agent program, which can then be deployed using an appropriately configured instance of the Agent Factory Run-Time Environment.

The AFAPL Inspector

Agent Factory provides support for the debugging of all types of agents via a generic debugging tool that is implemented as a run-time tool that can be deployed via the inclusion of a PLATFORM_GUI statement in the relevant platform configuration file. This tool is designed only for use with a single agent platform (other tools have been developed to provide multi-platform views), and provides support for inspecting the state of any agents or platform services that are resident on that platform. As a result, the tool currently provides two core views: an *agent view* and a *service view*.

The former view provides a list of agents together with a set of controls that can be used to start, stop and step individual or groups of agents. As expected, the start control resumes the execution of an agent, while the stop control suspends the execution of an agent. The step control is based on the step function that is offered by tradition debugging environments. Within AFAPL, a basic step is a single iteration of the AFAPL interpreter cycle. However, this has recently been extended to include support for breakpoints (Collier, 2006).

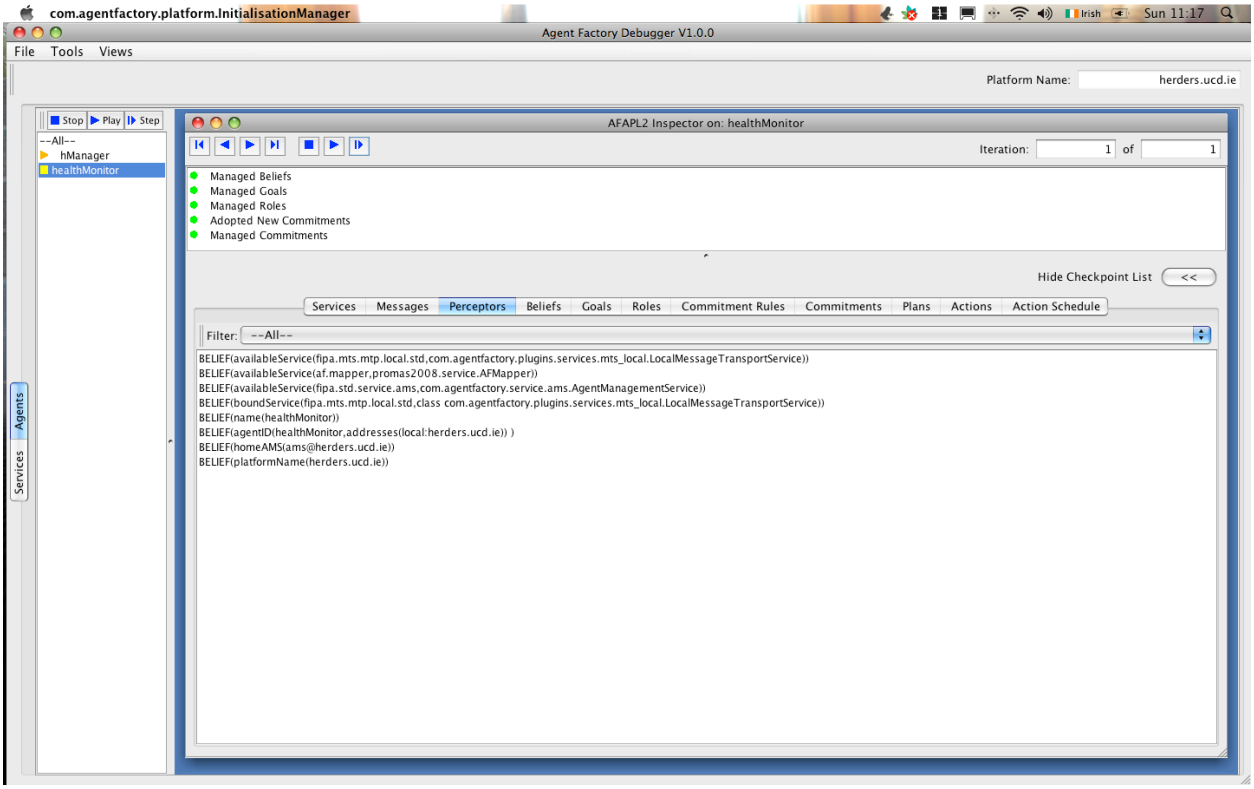

Figure 15: A Screenshot of the AF Debugger in which an AFAPL Inspector is being used to view the state of a Health Monitoring agent

Individual agent architectures / interpreters are then able to extend this basic snapshot to include By default, the Agent Factory Debugger includes an inspection tool that can be used to inspect the state of any type of agent. This is achieved by extracting the features that are core to all agent implementations, and includes support for the inspection of: all messages sent and received, and the platform services to which the agent is bound. This information is retrieved from a snapshot

object that can be generated by the agent architecture / interpreter on request. Specifically, on receipt of an end-of-time-slice message from an agent, the debugger requests a snapshot from that agent and stores the snapshot in a snapshot manager. This manager persists all previous snapshots to file and provides a mechanism for retrieving those snapshots on request. interpreter specific information, such as the current mental state of the agent. Visualisation of the snapshot is supported through the generic inspection tool (see Figure 15) that can be easily extended to include agent interpreter specific views.

For AFAPL, this inspector has been extended in two ways. Firstly, it has been modified to support the visualization of breakpoints, and secondly a number of additional views have been developed, such as:

- *Beliefs View*: Presents a view of the beliefs of the agent that can be customized either to show all beliefs or to show only the current beliefs, belief rules or temporal beliefs of the agent.
- *Commitments View*: Presents a view of the agent's current commitments. Selecting a commitment, allows the developer to view the commitment structure that is rooted at that commitment.
- *Commitment Rules View*: Presents a view of the agent's commitment rules. Selecting a rule allows the developer to see which of the currently active commitments were generated by that rule.
- *Action Schedule View*: Presents a view of the current state of the action schedule and a list of the actions that were completed in the last iteration of the interpreter cycle. This second list includes information about the time taken and the success or failure of the action.

Future Trends

Multi-Agent Systems has become a large research areas that encompasses a wide range of approaches, visions, and technologies. A recent Agent Technology Roadmap (Luck & McBurney & Shehory & Willmott, 2005), developed for AgentLink, the European Network of Excellence for Agent Research, outlined a number of issues and targets for the future of agent programming languages. These included a general lack of suitable tools, particularly for debugging and visualization; a lack of clarity in their semantics; missing concepts, such as support for organizational structures, social norms or models of cooperation; and a need for them to be evaluated through their use in the creation of large-scale agent systems.

As the roadmap document acknowledges, some of these issues have, at least partially, been solved already. However, there remain a number of outstanding issues. For example, little work has still been done on visualization of agent systems. Some visualization tools do exist, however they all tend to work with only one specific platform or toolkit. Further, many of these tools only work well for small collections of agents, and do not scale to larger systems. The lack of such support is significant as it make the development and debugging of large-scale agent systems more complex, if not impossible.

In addition to visualization, there is also a need to develop more comprehensive debugging tools. Traditionally, AOP researchers have focused on the development of inspection tools, such as the

one presented earlier. While these tools are easy to use when debugging simple agent systems, they do not scale well. Again, multi-layered support is required that combines offline log analysis tools [ref tracing tool], online monitoring tools, and the existing debugging tools. Additionally, there is a need to explore how other types of tool, such as profilers may be adapted to the agent domain.

From a program perspective, there is currently little consistency about the most appropriate set of features that underpin an agent system. This stretches from disparities in the choice of mental state to the way in which agents sense and effect their environment. These are further complicated by issues like the introduction of organizational concepts, such as roles and social norms; the need for features such as conversation management; and the incorporation of new work, such as argumentation.

Finally, given the autonomous adaptive nature of agent technology, it is vital that AOP languages begin to provide guarantees regarding the operation of the implemented agents. This ranges from guarantees of behaviour in uncertain environments to the integration of autonomic (Kephart & Chess, 2003) capabilities so that agents may continue to operate in the event of partial failure. Delivering such guarantees is vital is AOP languages are to be a viable option for new problem domains, such as distributed sensor networks, or control architectures for constellations of satellites / exploratory space vehicles.

Conclusions

In this chapter, we have presented an overview of a relatively new programming paradigm, known as Agent-Oriented Programming (AOP) (Shoham, 1993). AOP can be viewed as a specialization of Object-Oriented Programming in which objects are constrained to be agents, the state may only consist of mental attitudes, and message types are restricted to those provided by an underlying Agent Communication Language (ACL). In particular, the chapter has focused primarily on one AOP language, known as AFAPL, which supports the fabrication of agents that can be deployed through the Agent Factory framework.

AFAPL stands apart from many of the other current AOP languages by the fact that it has been extensively applied, over a prolonged period of time, to a wide range of non-trivial problem domains, such as mobile computing, robotics, virtual and mixed reality, and sensor networks. In fact, many of the enhancements made to both Agent Factory, and AFAPL have been as a direct result of experiences gained from work done in these areas. The primary exception to this is the JACK Agent Language (JAL) (Howden & Rönnquist & Hodgson & Lucas, 2001], which is part of JACK, a commercial agent development tool. Specifically, JACK has been used in the development of a number of commercial projects in a broad range of areas including defense, aerospace, telecommunications, manufacturing, and finance.

A common criticism of AOP languages has traditionally been the lack of tool support, and the custom nature of the tools that are provided. It is often argued that this leads to an increased learning curve for new developers; not only do they have to learn a new technology, but also they must learn how to use a new set of tools, and / or adapt existing tools to fill gaps in that tool set. One approach to reducing this learning curve is to integrate such tool support into existing

development tools. Unfortunately, this has not happened for many of the AOP languages. The two main exceptions to this are Jason (Bordini et al., 2005), the recent implementation of AgentSpeak(L) (Rao, 1996) and Agent Factory. As was described in the introduction to AOP, support for developing systems using Agent Factory has been provided in the form of a plugin for the Netbeans IDE. Similarly, Jason provides support for developing system via integration with the Eclipse IDE. JACK does provide an extensive toolset, however, this toolset is entirely custom built, and does not offer the range of support that is offered by more established IDEs such as Netbeans and Eclipse.

Another similar criticism of AOP languages is the lack of support for reuse. The ability to reuse code is often seen as vital to improving software development times. It allows developers to store generalized solutions to common problems that can be quickly adopted and adapted for different projects. An illustrative example of this is the Health Monitoring Service that is hinted at throughout the section on AFAPL. Such as service may be a common component of many implemented systems. Within AFAPL, a change in the design of this service requires updates in only two places: the HealthMonitor program, and the Monitored partial program. If no reuse mechanism is provided, then this change would need to be made to all agents that use the service. Currently, JACK and AFAPL are the only AOP languages that support code reuse.

Overall, the current state of AOP language research is promising. Many of the languages are continuing to develop and evolve as the state-of-the-art in Multi-Agent Systems research advances. Much ongoing work is being done on improving their robustness and expanding their feature sets to improve the expressiveness and simplify the task of creating more complex agent systems. This includes incorporating new concepts, such as roles and social norms, and expanding and improving the tool suites that are provided to support developers wishing to use an AOP language. That said, many of the languages now have a reasonably mature core set of constructs that are well defined and have been implemented robustly.

One of the main challenges facing AOP language developers over the next decade is to start to showcase the advantages that these languages offer over adopting a more traditional approach to agent development. This can only be achieved by using these languages to solve non-trivial problems and to highlight how adopting an AOP approach simplifies / improves the solution.

FIGURE CAPTIONS

REFERENCES

Bauer, B., Muller, J., Odell, J., (2001). Agent UML: A Formalism for Specifying Multiagent Interaction, *in P. Ciancarini and M. Wooldridge (Ed), Agent-Oriented Software Engineering, Springer-Verlag, 91-103.*

Bjorner D. and Jones, C., (1978). The Vienna Development Method: The Meta-Language. *Lecture Notes in Computer Science (LNCS 61), Springer Verlag.*

Bordini, R. H., Hübner, J. F., and Vieira, R. (2005). Jason and the Golden Fleece of agent-oriented programming. *In R. H. Bordini, Dastani, M., Dix, J., and El Fallah Seghrouchni, A., (Eds), Multi-Agent Programming: Languages, Platforms and Applications. Springer-Verlag, 3-37.*

Clynch, N. and Collier, R. (2007). SADAAM: Software Agent Development – An Agile Methodology. *in Proceedings of the Workshop of Languages, methodologies, and Development tools for multi-agent systems (LADS'007), Durham, U.K.*

Cohen, P. and Levesque, H. (1990). Intention is Choice with Commitment, *Artificial Intelligence 42, 213-261.*

Collier, R.W. (1996). The Realisation of Agent Factory: An Environment for the Rapid Prototyping of Intelligent Agents*, Master of Philosophy Thesis, University of Manchester Institute of Science and Technology (UMIST), England.*

Collier, R.W. (2001). Agent Factory: A Framework for the Engineering of Agent-Oriented Applications, *Doctoral Thesis, University College Dublin, Ireland*

Collier, R.W., O'Hare G.M.P., Lowen, T., Rooney, C.F.B., (2003). Beyond Prototyping in the Valley of the Agents, *in Multi-Agent Systems and Applications III: Proceedings of the 3rd Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'03), Prague, Czech Republic, Lecture Notes in Computer Science (LNCS 2691), Springer-Verlag.*

Collier, R., Rooney, C., O'Hare, G.M.P., (2004). A UML-based Software Engineering Methodology for Agent Factory, *Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering (SEKE-2004), Banff, Alberta, Canada.*

Collier, R., Ross, R., O'Hare, G.M.P., (2005). A Role-based Approach to Reuse in Agent-Oriented Programming**,** *AAAI Fall Symposium on Roles, an interdisciplinary perspective (Roles 2005), November 3-6, Hyatt Crystal City, Arlington, Virginia, USA.*

Dragone, M., Holz, T., O'Hare, G.M.P. (2007). Using Mixed Reality Agents as Social Interfaces for Robots, *Proceedings of the 16th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN 2007), Jeju Island, Korea.*

Duffy B.R., O'Hare G.M.P., Campbell A.G., Stafford J.W. & O'Grady M.J. (2005). NEXUS: Fusing the Real & the Virtual, *Proceedings of EUROCON 2005 - The International Conference on "Computer as a Tool", Belgrade, Serbia & Montenegro.*

Fikes, R.E., and Nilsson, N.J., (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving, *in Artificial Intelligence, 2, 189-208.*

Finin T., Labrou, Y., Mayfield, J. (1997). KQML as an Agent Communication Language. *In J. Bradshaw (Ed), Software Agents, MIT Press, Cambridge*

FIPA (2000). The Foundation for Intelligent Physical Agents Standards. Retrieved May 1, 2008, from http://www.fipa.org

Hindriks, K.V., de Boer, F.S., van der Hoek, W. & Meyer, J.-J.Ch. (1999). Agent Programming in 3APL. Autonomous Agents and Multi-Agent Systems, 2(4):357-401.

Howden, N., Rönnquist, R., Hodgson, A., & Lucas, A. (2001). JACK: Summary of an Agent Infrastructure, *Proceedings of the 5th International Conference on Autonomous Agents, Montreal, Canada.*

Kephart, J. O., & Chess, D. M. (2003). The Vision of Autonomic Computing. *Computer, 36(1), 41-50.*

Lillis, D., Collier, R., Toolan, F. and Dunnion, J. (2007). Evaluating Communication Strategies in a Multi-Agent Information Retrieval System, *Proceedings of the 5th European Workshop on Multi-Agent Systems (EUMAS'07), Hammamet, Tunisia.*

Luck, M., McBurney, P., Shehory, O., & Willmott, S. (2005). Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing). *AgentLink, ISBN 085432 845 9.*

Marsh, D., Tynan, R., O'Kane, D. & O'Hare, G.M.P. (2004). Autonomic Wireless Sensor Networks, *Engineering Applications of Artificial Intelligence Journal, 17(7), 741-748, Elsevier Science.*

Muldoon, C., O'Hare, G.M.P., Phelan, D., Strahan, R., Collier, R. (2003). ACCESS: An Agent System for Ubiquitous Service Delivery, *in Cooperative Information Agents VII, Lecture Notes in Computer Science (LNCS 2782), Springer-Verlag.*

McCarthy, J., (1979), Ascribing Mental Qualities to Machines, *M Ringle (Ed), Philosophical Perspectives in Artificial Intelligence, Harvester Press*

McCarthy, J. (1992), Elephant 2000: A Programming Language based on Speech Acts, *Unpublished Manuscript. Retrieved May 1, 2008 from http://www-formal.stanford.edu/jmc/*

O'Hare, G.M.P., Collier, R., Conlon, J. and Abbas, S. (1998). Agent Factory: An Environment for Constructing and Visualising Agent Communities, *9th AICS Conference, Irish Artificial Intelligence and Cognitive Science Conference, UCD, Dublin.*

O'Hare G.M.P., Duffy B.R., Collier R.W, Rooney C.F.B., O'Donoghue R.P.S., (1999). Agent Factory: Towards Social Robots, *First International Workshop of Central and Eastern Europe on Multi - agent Systems (CEEMAS'99), St. Petersburg, Russia*

O'Hare, G.M.P. & O'Grady, M.J. (2003). Gulliver's Genie: A Multi-Agent System for Ubiquitous and Intelligent Content Delivery. *Computer Communications, 26(11), 1177-1187.*

O'Hare G.M.P., O'Grady M.J., Marsh D., Ruzzelli A. G. & Tynan R. (2006). Autonomic Wireless Sensor Networks: Intelligent Ubiquitous Sensing, *Proceedings of ANIPLA 2006, International Congress on Methodologies for Emerging Technologies in Automation, University La Sapienza, Rome, Italy.*

Peng, L., Collier, R., Mur, A., Lillis, D., Toolan, F., and Dunnion, J., (2005). A Self-Configuring Agent-Based Document Indexing System. *Proceedings of the 4th International Central and Eastern European Conference on Multi-Agent Systems, Budapest, Hungary*

Powell, M.S. (1989), A program development environment based on persistence and abstract data types, *in I. Rosenberg and D. Koch (Eds), Persistent Object Systems, Springer Verlag.*

Rao, A.S. and Georgeff, M.P. (1991), Modeling Rational Agents within a BDI Architecture, *in J.Allen, R.Fikes, and E.Sandwall (Eds), Proceedings of Second International Conference on Principles of Knowledge Representation and Reasoning, 473-484, Morgan-Kaufmann, San Mateo, CA.*

Rao, A.S., and Georgeff, M.P. (1995). BDI Agents: from theory to practice, *in Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95), 312-319, San Francisco, CA, June.*

Rao, A. (1996), AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language, *Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world, Einhoven, Netherlands, 42-55.*

Robinson, J. A. (1965). A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM 12, pp23- 41.*

Rooney, C.F.B., Collier, R.W., & O'Hare, G.M.P. (2004). VIPER: VIsual Protocol EditoR. *Proceedings of the 6th International Conference on Coordination Models and Language (COORDINATION 2004), Pisa, Italy, 24-27 February.*

Ross, R., Collier, R., & O'Hare, G.M.P., (2004). AF-APL – Bridging Principles & Practice in Agent-Oriented Languages. *Proceedings of the 2nd International Workshop on Programming Multi-Agent Systems Languages and Tools (PROMAS- 2004), New York.*

Searle, J.R. (1969), Speech Acts, *Cambridge University Press, Cambridge, UK.*

Shoham, Y. (1993), Agent-oriented programming, *Artificial Intelligence, 60(1):51-92*.

Thomas, S. R. (1995), The PLACA Agent Programming Language, *in M.J. Wooldridge and N.R. Jennings (Eds), Intelligent Agents, Lecture Notes in Artificial Intelligence (LNAI-890), 355-370, Springer-Verlag.*

Wooldridge, M. (2000), Reasoning About Rational Agents, *The MIT Press, Cambridge, MA*.

Wooldridge, M. & Jennings, N.R. (1995), Intelligent Agents: Theory and Practice, *Knowledge Engineering Review* 10(2).