Towards Pervasive Intelligence: Reflections on the Evolution of the Agent Factory Framework

C. Muldoon, G. M. P. O'Hare†, R. W. Collier, M. J. O'Grady

Abstract Agent Factory is a cohesive framework for the development and deployment of multi-agent systems. Since its inception in the mid 1990s, Agent Factory has gone through a metamorphosis process, whereby several new extensions, revisions, and enhancements have been made. This chapter provides a discussion of the incremental developments in Agent Factory and provides motivations as to why such changes were necessary. Agent Factory distinguishes itself from other intentional agent platforms in several ways. It provides a practical and efficient approach to the development of intentional agent-oriented applications. This is combined with a methodology, integrated development environment support, and a suite of tools that aid the agent fabrication process. A detailed comparison to related work is provided. We include a tutorial on how to use the framework.

1 Introduction

A curtsey examination of the research literature will quickly indicate the interest that Multi-Agent Systems (MAS) attract from the Distributed Artificial Intelligence (DAI) community. A number of frameworks, some well-known, others less so, have been documented since the mid 1990s. Agent Factory is an exemplar framework in that it was conceived of and developed at this time, but it has been significantly enhanced over the years in response to ongoing software developments. For example, Wireless Sensor Networks (WSNs) offer a promising solution for many categories of prob-

†Corresponding Author

C. Muldoon, G. M. P. O' Hare, R. W. Collier, and M. J. O'Grady
CLARITY: The Centre for Sensor Web Technologies, School of Computer Science & Informatics, University College Dublin, Belfield, Dublin 4, Ireland, e-mail: {conor.muldoon,gregory.ohare,rem.collier,michael.j.ogrady}@ucd.ie

lems, such as real-time environmental monitoring, however, such networks are characterised by limited computational resources. An optimised Agent Factory runtime environment has recently been developed that factors these issues into its design. Thus an Agent Factory MAS can be deployed on a wide range of hardware, extending and increasing the application domains that can harness the MAS paradigm. Today, Agent Factory represents the culmination of over a decades effort by a number of researchers and has been successfully demonstrated in a wide range of application domains.

This chapter will discuss various facets of Agent Factory and its constituent components. In Section 2, a motivation of Agent Factory is presented. This section discusses the chronological enhancements/alterations made to the system over the course of its development. Section 3 discusses the Agent Factory Agent Programming Language (AFAPL), AFAPL is an agent-oriented programming language that is based on a logical formalism of belief and commitment. The agent platform is discussed in Section 4. At present, Agent Factory is divided, as with many other frameworks, into two editions, one for standard Java, the other for Java Micro Edition (JME) CLDC. We describe some applications of the framework in Section 5. Finally, a brief but succinct comparison of Agent Factory with other frameworks is presented in Section 6.

2 Motivation

In order to motivate the Agent Factory platform and language, the Agent Factory Agent Programming Language (AFAPL), we must put the current system into context; as such, in this section, we provide the history of the system and discuss its evolution into its current form. This provides several motivations as to why certain design decisions were made and reflects the general changing trends in both agent technology and software development in general over the past decade. For instance, when the initial version of the system was developed, the now pervasive Java programming language was only being released and was still a very new and unproven platform. We discuss Agent Factory's development in a chronological narrative that begins in the mid 1990s and progressively introduces the main features of the language and framework, ending with a overview of the system in its current state.

2.1 History

Agent Factory was first proposed by O Hare in 1996 [39] and was conceived of as an environment for the prototyping and testing of multi-agent systems.

It supported the rapid creation of central agent components and the aggregation of such components into communities, whose model of social interaction would subsequently be described. It differed from other similar frameworks available at the time in its use of an algebraic specification technique (the UMIST Paradox System) that could be directly executed. The system was underpinned by the theory associated with heterogeneous algebras that was developed by Birkhoff and Lispson [5]. The environment drew heavily from Communicating Sequential Processes [23] (CSP) and viewed multi-agent systems as a specific class of complex distributed systems, namely Communicating Intentional Processes (CIP). It built upon and extended pre-existing work to model CIPs effectively [40].

Early work on Agent Factory extended the framework to include an agent classification hierarchy to aid the rapid prototyping of agent designs and a communication infrastructure, based on TCP/IP and Speech Act theory, that enabled the agent communities to be distributed over a network [9]. Due to the difficulties of using the original algebraic approach, this version of the system was implemented using a combination of VDM and Smalltalk-80. The main reason for using Smalltalk-80 was that it was a flexible object-oriented language and thus provided distinct advantages, in terms of rapid prototyping, maintainability, and ease of development, over attempting to directly execute the specifications. At the time, Java was only being released, and was quite a new and unproven platform. C++ was widely used at this time, but it was (and still is) viewed as a bastardised language rather than truly object-oriented. Additionally, Smalltalk contained features, such as reflection, not supported in C++.

Further work redesigned Agent Factory to reflect newer trends in agent development [11]. A generalized agent communication language framework was introduced along with a commitment management system module that could be configured for different commitment models and/or strategies. A mobility feature was subsequently introduced [13]. Up to this point, much work had been done in the field on mobility, but it primarily focused on the actual mechanism for agent transfer; the intelligent component and decision making process in migration was largely ignored. Agent Factory merged the notion of intentional decision making with mobility.

At the start of the millennium, work began to focus on the possibility of using Agent Factory for the creation of mobile context sensitive applications, such as Guilliver's Gennie [38]. A new Java version of the system was developed. The reason for this development was that, at the time, there were no Smalltalk-80 interpreters or byte code compilers/translators for the Microsoft Pocket PC operating system. There was, however, a (rather slow) Esermertec Jeode Java interpreter that was based on PersonalJava, the forerunner to what is now known as the JME CDC Java configuration augmented with Personal Profile. At this time, there were two versions of Agent Factory in operation. The Smalltalk-80 version was intended for desktop environments, whereas the Java version was intended for mobile devices. This motivated the

development of the Agent Factory Agent Programming Language (AFAPL), which drew heavily from Shoham's AGENT0 [44] language and Agent Oriented Programming. AFAPL was an abstract declarative language that enabled agents to be created that were based upon a theory of rational agency. It enabled the creation of agent designs that were independent of the underlying imperative environment in which they were to operate. The idea was that different interpreters, written in different languages, could be developed for AFAPL, but the specification of the agent design would remain consistent regardless of the environment in which it would subsequently be executed. Additionally, AFAPL supported the notion of agent design inheritance. This enabled the creation of generic agent designs that could be reused and augmented with application specific functionality. This version of the Agent Factory laid the foundation for Collier's thesis [10], which provided a cohesive framework for mobile distributed multi-agent application development, and was based upon a logical formalism of belief and commitment. The thesis introduced an agent development methodology that drew from early work by O Hare and Wooldridge [41] on Agent-Oriented Software Engineering.

Due to the differences in the languages and the need to duplicate new features of the system, it was decided to drop the Smalltalk-80 version of the system. It had to be the Smalltalk version of the system that was dropped because, as noted earlier, there was no Smalltalk-80 interpreter for Pocket PC. PersonalJava was, more or less, a full Java platform for embedded devices, almost identical to standard Java less the Swing graphical interfacing capabilities. It was therefore possible to have a single Agent Factory platform for mobile and desktop applications. The Smalltalk-80 version of the system didn't add enough to justify its continued maintenance.

At this time, the primary general purpose programmable mobile devices being used were Personal Digital Assistants (PDAs). This market, however, never really took off in the mainstream. Much of the functionality of the PDA was beginning to be incorporated in to mobile phones. In contrast to the PDA market, the mobile phone market was in the billions. The problem, however, was that the standard Java environment for phones was the J2ME Constrained Limited Device Configuration (CLDC) augmented with the Mobile Information Device Profile (MIDP). Agent Factory was developed for use with the CDC Java environment augmented with Personal Profile. This environment, although classified as J2ME, is closer to standard Java than it is to CLDC, which is highly constrained and limited. Therefore, it was not possible to use Agent Factory, in its current form, for the development of mobile phone applications.

In 2005, Muldoon developed a minimised footprint version of Agent Factory that facilitated the development and deployment of intentional agents for highly constrained mobile and ubiquitous devices [35, 34, 33]. This version of Agent Factory was referred to as Agent Factory Micro Edition (AFME). Initially, AFME targeted cellular digital mobile phones. It soon became clear, however, that AFME could be deployed on a much broader array of devices,

since it was based on CLDC, which is the de factor standard Java environment for constrained environments. In 2007, Sun launched the SunSPOT mote in Europe, which was based on CLDC. AFME was subsequently deployed on the SunSPOT , making it the first intentional agent platform to be used to control the leaf nodes of a Wireless Sensor Network (WSN). AFAPL was developed to provide a common high-level language that would be consistent between Smalltalk-80 and Java. Although Smalltalk-80 was no longer supported, this approach proved useful again, but in a different context, to ensure that an agent could be represented consistently among different Java environments.

In parallel to the development of AFME, a new concept of a role was introduced to AFAPL [8]. This version of AFAPL became known as AFAPL2. The notion of a role improved the efficiency of the platform. The idea was that, at various stages throughout execution, triggers would cause commitment rules to be added to the rule set of the agent. That is, an agent would adopt various behaviours, referred to collectively as a role, if some trigger condition was true. The trigger condition was evaluated by checking the agent's beliefs. If an agent had a particular belief and that belief matched the trigger condition, the role would be adopted. If the role was no longer relevant or the objectives associated with the role were achieved, the role would be retracted by an actuator. In this way, the overhead for evaluating the role would only be incurred at times when the role was active. The trigger did not represent a maintenance condition for the role. The execution of the retraction actuator would be evaluated in the usual manner i.e. either directly though the evaluation of the commitment rules or through the execution of a plan. The version of Agent Factory represents its current form.

3 Language

Agent Factory is an open framework for building agent-based systems (see Section 4), and as such, does not enforce a single flavour of agent upon the developer. Instead, the developer is free to either use a pre-existing agent interpreter / architecture, or develop a custom solution that is more suited to their specific needs.

That said, as was highlighted in Section 2, Agent Factory has a long tradition of promoting the use of intentional agents to construct multi-agent applications. Support for this style of agent has been realized through the Agent Factory Agent Programming Language (AFAPL), which is now in its second incarnation, and is known as AFAPL2. AFAPL2 is an agent programming language that supports the development of agents that use a mental state architecture to reason about how best to act. The remainder of this section will provide a brief summary of AFAPL.

The AFAPL2 language supports the fabrication of agents whose mental state is comprised of beliefs, goals, and commitments. Beliefs describe - possibly incorrectly - the state of the environment in which the agent is situated, goals describe future states of the environment that the agent would like to bring about, and commitments describe the activity that the agent is committed to realising. The behaviour of the agent is realised primarily through a purpose-built execution algorithm that is centred about the notion of commitment management.

Commitments are viewed as the mental equivalent of a contract; they define a course of action/activity that the agent has agreed to, when it must realise that activity, to whom the commitment was made, and fnally, what conditions, if any, would lead to it not having to fulfil the commitment. Commitment management is then a meta-level process that AFAPL2 agents employ to manipulate their commitments based upon some underlying strategy known as a commitment management strategy. This strategy specifies a set of sub-strategies that:

- define how an agent adopts new commitments; maintains its existing commitments.
- refines commitments to plans into additional commitments.
- realises commitments to primitive actions; and handles failed commitments.

The principal sub-strategy that underpins the behaviour of AFAPL2 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 commitment. The adoption of a primary commitment occurs as a result of one of two processes: (1) in response to a decision to attempt to achieve a goal using a plan of action, or (2) as a result of the triggering of a commitment rule. Commitment rules define situations (a conjunction of positive and negative belief atoms) in which the agent should adopt a primary commitment.

3.1 Specifications and Syntactical Aspects

The first step in understanding how to program AFAPL2 agents is to understand how beliefs are used to construct models of the current state of the environment. Without this model, the agents will not be aware of what is happening in the environment, and consequently, will not be able to act in a meaningful way.

The key step underpinning the generation of an agents' belief model is perception. This is the process by which an agent converts raw environment data (sensor readings, ACL messages, address books, etc.) into various beliefs

that provide a higher-level representation of this data (and consequently, the state of the environment). Perception is an integral part of the belief update phase of the AFAPL2 interpreter cycle.

The principle building block of the perception process is the perceptor unit. This is a Java class that collates any relevant raw data and generates a corresponding set of beliefs. Perceptors are associated with agents via the PERCEPTOR construct. This construct generates a mapping between specific perceptor units and a given AFAPL2 agent program. For example, when developing a robot soccer agent, a ball perceptor unit may be created that uses visual information to decide whether or not the robot has the ball. In the AFAPL2 program, this perceptor would be declared through the following statement:

```
PERCEPTOR BallWatcher  CLASS perceptor.BallPerceptor;
```

Where perceptor.BallPerceptor is the Java class that implements the perceptor unit. This unit would then be responsible for generating a corresponding belief about the presence of the ball (e.g. BELIEF(hasBall) or BELIEF(noBall)).

AFAPL2 has been designed to support the fabrication of agents that exist within highly dynamic environments. As such, agents may adopt beliefs that quickly become invalid. For example, consider a robot soccer agent that has a perceptor which generates a belief about whether the agent can see the ball or not. If the ball passes quickly in front of the agent, then it may see the ball only for one or two iterations of the interpreter cycle.

Rather than implement a complex belief revision algorithm that tries to understand when a belief has become invalid, the approach adopted in AFAPL2 is to assume that, by default, all beliefs become invalid at the end of a given iteration of the AFAPL2 interpreter cycle. In this way, perception becomes the process by which the agent generates a snapshot of the current state of the environment. This snapshot is then thrown away immediately before the next snapshot is generated.

While this approach helps to simplify the maintenance of an agents beliefs, it is not always appropriate (sometimes we need beliefs that persist for longer). To handle this requirement, AFAPL2 also provides a number of temporal operators, which can be used to define beliefs that persist for more than one iteration.

This first example illustrates how to create an Agent Factory perceptor. To implement a perceptor, we create a Java class that is a subclass of the com.agentfactory.logic.agent.Perceptor class, and implement the perceive() method:

```
import com.agentfactory.logic.agent.Perceptor;
```

```
public class AlivePerceptor extends Perceptor {
   public void perceive() {
       adoptBelief( "BELIEF(alive)" );
   }
}
```

The above perceptor generates a single belief that represents the fact that the agent is "alive". This belief is added to the agent's belief set at the start of each iteration of the interpreter cycle. The name of the Java class is specified within the agent design file.

The next example illustrates how to specify an action (and the corresponding actuator) in AFAPL2. Actions are used to enable agents to affect their environment. To implement an action, we must do two things: first, we need to create an actuator that contains the implementation of the action. After this, we need to specify our action in an AFAPL2 file, using the ACTION construct.

Actuators are Java classes that subclass the com.agentfactory.logic.agent. Actuator class and implement the act(..) method. Upon creation, the agent creates on instance of each specified actuator. Thus, the same instance is used even when the action is being performed concurrently. Consequently, actuators must be implemented as thread-safe classes.

To illustrate how to create an actuator, we will develop a "helloWorld" action that prints the string "Hello World" to the console.

```
import com.agentfactory.logic.agent.Actuator;
import com.agentfactory.logic.lang.FOS;

public class HelloWorldActuator extends Actuator {
public boolean act( FOS action ) {
System.out.println( "Hello World" );
return true;
    }
}
```

What the above actuator implementation does is fairly obvious. The only "issue" is the return value of an actuator. This is used to define whether the commitment to the corresponding action failed or succeeded. This is useful in situations where it is possible for the actuator to complete unsuccessfully, for example, updating a table in a database. In such cases, the actuator can indicate its failure by returning false instead of true.

For an actuator to be used by an agent, an action definition for the actuator must be specified in the agent design. The action definition below specifies an action called "helloWorld" and links the action to the HelloWorldActuator. In addition, this definition requires that any pre and post conditions that apply to the action be specified.

Pre-conditions are used to ensure that the action is only performed when it is possible. For example, a robot soccer agent program may include a "kick" action. The pre-condition for this action would be that the agent has the ball (i.e. BELIEF(hasBall)). Conversely, post-conditions are used to identify which actions can be used to achieve the goals of the agent. For this example, we will declare both the pre- and post- condition of our action to be true (this is a default that means "no precondition or postcondition").

```
ACTION helloWorld  PRECONDITION BELIEF(true); POSTCON-
DITION BELIEF(true);
  CLASS helloworld.HelloWorldActuator;
```

Many scenarios require that an agent act in response to some change in its environment (for example the receipt of a message from another agent, the triggering of a sensor, the location of additional resources for processing, and so on). In such cases, we require a mechanism that allows the developer to define situations in which the agent must commit to some activity (i.e. some action or plan). Agent Factory supports this through the use of commitment rules. Commitment rules specify situations (defined as a belief sentence) in which the agent should adopt a commitment.

The following example illustrates the situation in which we want to program our robot soccer agent to move towards the ball whenever it sees it. This is achieved through a rule of the following form:

```
BELIEF(seesBall)  =>  COMMIT(?self,  ?now,  BELIEF(seesBall),
moveTo(ball));
```

This rule states that, if the robot soccer agent sees the ball, then it should commit to the action of moving to that ball. Two key points to take from the above example are:

- The introduction of two pre-defined variables, ?self and ?now, which are bound to constants representing the agents name and the current time respectively.
- The use of a maintenance condition to constrain the persistence of the commitment when adopted. The agent maintains the commitment to move to the ball until either the moveTo(ball) action completes or the agent no longer believes that it sees the ball.

Should the robot soccer agent ever come to believe that it sees the ball (i.e. it has the belief BELIEF(seesBall)), then the commitment rule would be fired. This would cause the agent to adopt the corresponding commitment. So, if the agent was called striker, and it saw the ball at 11:46am, then it would adopt a commitment of the form:

```
COMMIT(striker, 11:46, BELIEF(seesBall), moveTo(ball))
```

The above commitment rule specifies a behaviour that is realised through the adoption of a single commitment. Commitment rules can also be used to drive the adoption of multiple commitments simultaneously. This can be achieved by introducing variables into the situation definition.

For example, consider an agent-based security system that includes a monitoring agent that is responsible for monitoring what Radio Frequency IDentification (RFID) tags that enter or leave a specified region (which is monitored by one or more RFID antenna). This agent may be designed to handle beliefs of the form BELIEF(newTag(?tagID)) where ?tagID is a unique code that is assigned to every RFID tag, and the belief itself is used to represent the fact that an new RFID tag has entered the monitored region.

The expected behaviour of this agent is that it will perform a security check whenever a tag enters the monitored region. The agent uses the result of the security check to determine whether or not it should raise an alarm.

To implement this behaviour within AFAPL2, a commitment rule of the form is added:

```
BELIEF(newTag(?tagID)) => COMMIT(?self, ?now, BELIEF(true),
checkTag(?tagID));
```

Informally, this rule states that, if the agent detects that a new RFID tag has entered the monitored region, then it should perform a check to see whether that tag is allowed to be in the monitored region. What the agent does when the tag has been checked can be specified through the introduction of additional commitment rules. For example:

```
BELIEF(illegalTagMovement(?tagID)) & BE-
LIEF(tagAuthority(?agentName, ?agentAddress)) => COMMIT(?self,
?now, BELIEF(true), inform(agentID(?agentName, ?agentAddress),
illegalTagMovement(?tagID));
```

This second rule states that if the agent believes that a tag is not allowed to be in the monitored region (this is the first of the beliefs on the left hand side of the belief rule) and it knows a tag authority agent (this is the second of the beliefs on the left hand side of the belief rule), then it informs the tag authority agent that it has detected an illegal tag movement (this happens through the adoption of the commitment on the right hand side of the commitment rule).

Agent Factory uses resolution-based reasoning to evaluate the truth of the belief sentence part of a commitment rule. To illustrate this point, let

us consider the RFID scenario in more detail. The agent is responsible for monitoring the movement of objects in a physical space of a building using RFID tags. In such systems, the actual monitoring of the space is carried out by an one or more RFID antenna. The corresponding agent is then linked to that antenna (or set of antenna) via some form of interface that generates events when RFID tags enter or leave the monitored space. To make the agent aware of these events, we introduce an event perceptor that generates beliefs based on the events that are produced by the interface. For events where an object that has an RFID tag enters the monitored space, the perceptor generates beliefs of the form BELIEF(newTag(?tagID)), which corresponds to the belief on the left hand side of the second commitment rule.

As an example, consider the case where a single tagged object (with a unique identifier of 101 - for simplicity) enters the region that is monitored by an agent with identifier "lobby". The entry of this tag is detected by the antenna and passed to the agent's perceptor via the interface. This causes the perceptor to generate the belief BELIEF(newTag(101)). The adoption of this belief causes the second commitment rule to be triggered. That is, the belief sentence on the left hand side of this commitment rule is evaluated to true when the variable binding ?tagID / 101 is applied. This results in the adoption of a single commitment of the form:

COMMIT(lobby, 9:28, BELIEF(true), checkTag(101))

If, at the same time, a second tag, with identifier 320, also entered the monitored region, then the agent would have a second belief of the form BELIEF(newTag(320)). This would cause the query process to generate two variable bindings for the second commitment rule: tagID / 101 and ?tagID / 320 . Based on these bindings, two commitments would now be adopted by the agent: the commitment above, and a second commitment of the form:

COMMIT(lobby, 9:28, BELIEF(true), checkTag(320))

So, what this example highlights is that the interpreter generates every possible variable binding for the belief sentence component of each commitment rule. These bindings are then applied to the commitment component of each commitment rule and the resultant commitments are adopted.

To summarise some of the main features of the language:

- The language is deliberative and is based on a logical formalism of beliefs and commitments. Reactive behaviour within the framework is encoded imperatively within Java (more recently, some work has been done to put structure around the reactive behaviour within the SoSAA architecture [18]).

- The language provides high level speech act primitives for communication. It facilitates the broadcast of messages through the use of wild card pattern matching.
- The language is relatively easy to understand for developers that have experience of declarative or logic-based programming. For developers without such experience, the learning curve is steeper.

3.2 Semantics and Verification

The language has a clear and precise semantics. The original version of AFAPL was formalised in Collier's Thesis [10]. Work is underway to provide a formal operational semantics of the language in its current form for a future paper. The a language is suitable for the creation of a wide variety of agent-oriented applications (see Section 5).

3.3 Software Engineering Issues

The Agent Factory system, in general, has been designed with Software Engineering concerns at its core. Developing a system and methodology that had a sound software engineering basis was one of the contributions of Collier's thesis [10]. The Agent Factory system is modular and can be extended with functionality for a number of different agent architectures. The design of AFME has been strongly influenced by good objected oriented precepts, such as the 'Law of Demeter' (LoD) [30]. The LoD, which was popularised by Grady Booch and James Rumbaugh [6, 43] in the 1990s, greatly improves the maintainability of the software. As with many researchers in the Software Engineering and Object-Oriented community, we view the overuse of inheritance as a bad thing and tend to favour composition.

The language is integrated with Java. At present, no support is provided for integrating the language with other classical (imperative) languages, but there is no reason why such support could not be provided in the future.

3.4 Other features of the language

The platform supports the construction and deployment of mobile agents. Truly strong migration is not possible in Java because much of the system state is controlled by the JVM and cannot be directly accessed (see Section 4). Within Agent Factory support is provided for the migration of both code (application classes) and the agents' mental state. Within AFME, support

is only provided for the transferral of state. The language does not facilitate the definition of new language components from basic constructs.

4 Platform

As with many other agent frameworks (see Section 6), Agent Factory has been divided into two editions, one for standard Java, the other for JME CLDC. In this section, we initially discuss the standard platform, then AFME.

Agent Factory is an open source framework that has been developed to support the deployment of multi-agent systems [12]. 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. As such, the framework has become an enabling middleware layer that can easily be extended and adapted for different application domains.

The Agent Factory framework can be 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 that have been built to provide infrastructure services for applications;
- 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.

The critical components used in the deployment of a multi-agent system are the agent platforms and the development kits. Agent platforms deliver the machinery necessary to create and support agents that are developed using one or more of the development kits. These development kits include:

- an agent container that holds the agents currently resident on the agent platform;
- a platform service 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;
- an architecture factory module that manages the instantiation of the various agent interpreters / architectures that can be deployed 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.

Conversely, each development kit provides some form of template agent interpreter / architecture that can be used to create application agents. The most common support tools that a development kit provides are: a customized agent inspector that allows the visualization of the internal state of that architecture; and code templates for an appropriate IDE. However, other support tools may be added if deemed appropriate. Currently, Agent Factory provides two basic development kits: the AFAPL2 Development Kit, which supports the creation of intentional agents; and the Reactive Message Agent (RMA) development kit, which provides a simple reactive agent architecture that combines message handlers, event handlers, and an internal memory.

The configuration of the agent platform and the specification of the initial agent community are supported via two deployment files: the platform configuration file is used to specify a configuration for each agent platform and the agent platform script is used to specify the initial community of agents that are to be deployed on that platform.

The platform configuration file aids developers in defining what platform services, agent architectures, and run-time tools should be deployed on each agent platform. It is also used to assign a default name to each agent platform. Perhaps the most common use of this configuration file is to specify what message transport services will be made available to agents residing on the platform. They are deployed as platform services, and currently, implementations exist for local message passing, HTTP, UDP and XMPP.

Conversely, the agent platform script is used to declare the initial agent community that will be deployed on an agent platform at start-up. This file allows developers to define what agents will be created, to initialize the state of those agents, and when 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. 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 do not.

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 (AFAPL).

4.1 Agent Factory Micro Edition

Agent Factory Micro Edition (AFME) is a minimized footprint version of the Agent Factory platform designed specifically for use with resource constrained devices. AFME was originally used for the development of applications for cellular digital mobile phones, but has since been ported to the leaf nodes of a WSN and specifically Sun SPOT motes. In deploying agents on sensor nodes, developers are faced with a number of problems; perhaps the most obvious is the limited spatiotemporal computational and power resources available. It is for this reason that initial agent frameworks developed for WSNs were based on a weak notion of agency, whereby agents did not possess reasoning capabilities, the canonical example being Agilla. Other more recent approaches focus on particular algorithms for agent interaction [46], but the agents would not be considered strong in the traditional sense of the word.

This chapter advocates the use of AFME agents, which are loosely based on the BDI notion of agency. The BDI model is an appropriate paradigm for devices such as WSN motes in that it acknowledges that agents are resource constrained and will be unable to achieve all of their desires even if their desires are consistent. An agent must fix upon a subset of desires within its intention selection process and commit resources to achieving them.

Traditionally, the BDI model of agency would be considered too computationally intensive for small devices, such as WSN motes. In this paper, we agree with this argument, in so far as that we are not proposing to deploy BDI agents on extremely low specification nesC type devices, such as the Berkeley or Tyndall motes. Nonetheless, there are now several devices on the market, such as the Sun SPOT or Imote2, that have considerably more resources available than early motes. On such devices, it is quite feasible to deploy minimized footprint BDI agents. For this to be practical, however, it is still essential to ensure that resources are not squandered and are used in an intelligent and prudent manner. BDI languages are, for the most part, declarative although in practice most of them are used in conjunction with imperative components. In general, the declarative components specify a theory of the problem to be solved, whereas as the imperative components specify the low level coding procedures. The imperative components are usually written in object languages, such as Java or C++. The agent languages represent a logical abstraction. Various interpreters can be built for them so that they can be used in different environments. There are no failsafe development methodologies that ensure a good agent design. The design decisions made are of significant importance in the WSN domain where resources are extremely scarce.

There is an inherent cost in controlling a system and in performing computations. The complexity of algorithms puts certain constraints on the time in which we may obtain the results of the computation; therefore there is a limit to the amount of knowledge or information we can attain at a particular time point. Either we accept errors due to the lack of information and allow

the system to be responsive or we allow the system to carry on operating in a suboptimal manner as we are performing computations or obtaining information so as to make better decisions. That is, the fact that performing a computation has a spatiotemporal and energy usage overhead prevents us from controlling a large system perfectly [3]. In developing software systems in practice, this ultimately comes down to a granularity problem.

In AFME, we provide the developer with functionality to encode the deliberative behaviour of agents yet they may also encode functionality at an imperative level directly within Java. The decision as to whether a particular task should be declarative or imperative is not clear cut and ultimately comes down to the experience and knowledge of the developer. It often depends on whether the developer believes something should be a task or a goal. Tasks are less expensive in that they use fewer resources and the result may be obtained faster, but they are also less flexible and reduce the possibilities for recovery from failure.

In developing BDI agents for sensor networks, the developer could be tempted to develop everything as a task, but this would be little better than using the weak approach to agency. At the other extreme if the agent does too much reasoning, resources will be wasted and the system will be unresponsive. The development framework discussed in this article supports both approaches. It is our belief that it is no longer the case that the BDI model of agency is too computationally intensive for resource constrained devices. With developments in computing technology, improvements in the efficiency of algorithms, and the dissemination of good design practices and the knowledge of algorithms to developers, the traditional arguments no longer hold. It is for this reason that a number of agent-based ambient systems have begun to emerge. To the best of our knowledge, AFME was the first BDI framework to be deployed on the leaf nodes of a wireless sensor network.

Each agent in AFME consists of a set of roles, which are adopted at various points throughout execution. Each role consists of a trigger condition and a set of commitment rules. Once an agent adopts a belief that matches the trigger, the role is adopted and the set of commitment rules are added to the agent's mental state. Subsequently, on each iteration of the agent's control process, the commitment rules are evaluated until either the role is retracted or the agent is terminated. The set of commitment rules adopted when a role is triggered specify the conditions under which commitments are adopted for the role. Originally, these conditions only included the agent's beliefs, but more recently, in AFME, support has been added for equalities, inequalities, and rudimentary mathematic operations. This is useful because it allows developers to specify, at a declarative level, relationships among beliefs. For instance, if an agent had beliefs about the cost of bread and butter, the developer could encode conditions such as if bread costs more than butter or if bread costs less than butter minus 10. With the original approach, this is not possible without writing imperative code to compare the beliefs or belief arguments. Once commitments have been adopted, the agent commences

the commitment management process. Various arguments are passed to the commitment when it is adopted, such as the time at which it should commence, to whom the commitment is made, and the maintenance condition of the commitment. An identifier is specified which acts as a trigger for the plan or primitive action to be performed. In subsequent iterations of the control algorithm, the commitment is invoked subject to the arguments specified.

4.1.1 AFME Platform and Life Cycle

An AFME platform comprises a scheduler, several platform services, and a collection of agents. The scheduler is responsible for the scheduling of agents to execute at periodic intervals. Rather than each agent creating a new thread when they begin operating, agents share a thread pool. Platform services are a shared information space amongst agents. Platform services, such as the local message transport service, are required in situations where agents must gain access to a shared object instance so as to act upon the object or perceive its state.

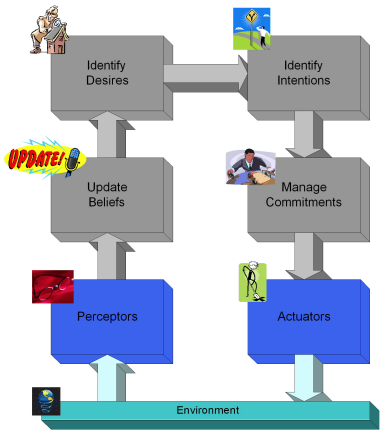

Fig. 1 The AFME Control Process

AFME delivers support for the creation of agents that follow a sense-deliberate-act cycle. The control algorithm performs four functions (see Figure 1). First, preceptors are fired and beliefs are updated. Second, the agent's desired states are identified. A subset of desires (new intentions) is chosen, and added to the agent's commitment set. It should be noted that if the agent has older commitments, which are of lower importance, they will be dropped if there is not enough resources available to execute all commitments. This is handled through the knapsack procedure. Fourth, depending on the nature of the agent's commitments, various actuators are fired.

4.2 Available tools and documentation

Both Agent Factory and AFME are well documented and provide tool based support for the development and debugging of applications:

- The platform has extensive documentation available from the Agent Factory website (http://www.agentfactory.com). This includes tutorials on how to use the framework and technical documentation, such as Javadoc.
- An agent logger and visual debugging tool, which enables the developer to examine the agent's mental state, is available.
- On-line help is available from the website along with information on how to download and install the system.
- Tools for the administration, management, and configuration of the platform are integrated into the Netbeans IDE.

4.3 Agent Factory/AFME Integration

Agent Factory and AFME are integrated through the message transport and migration services. Both platforms are FIPA compliant therefore and message transfer and communication is consistent. In terms of migration, it is necessary for agents to change their form when migrating from a standard desktop environment to a constrained device and vice versa. The remainder of this section will discuss how the message transport and migration services of AFME have been developed to integrate with Agent Factory.

4.3.1 Message Transport Service

The Message Transport Service of AFME had to be changed considerably from the original design. This was because the local GPRS and 3G service providers have a firewall operating to prevent incoming socket connections and also because MIDP and J2SE support different APIs for networking. Rather than having a server operating on the mobile device, the message transport service periodically polls a mailbox server operating outside the firewall domain. Incoming messages are stored in the mailbox until a connection is made from the client devices, at which point all stored messages are transferred. This increases the latency of message passing but is necessary to pierce the firewall.

The message transport service has two modes of operation, namely synchronous and asynchronous. These modes are related to how outgoing messages are processed. Incoming messages are handled generically. When operating in synchronous mode, all outgoing messages are buffered within the service. When a connection is made to the mailbox server and all incoming

messages have been received, the same socket connection is kept open and used to transfer outgoing messages. When the mailbox server receives the outgoing messages, they are forwarded on to their destination. When operating in asynchronous mode, outgoing messages are sent directly within their own individual sockets. This is possible because the firewall only blocks incoming sockets not outgoing. The choice made of which mode to use when developing an application depends on whether the developer wishes to minimize latency or maximize performance. When operating within synchronous mode, there will be less socket connections made, whereas in asynchronous mode the latency of outgoing messages will be lower.

When the mailbox server receives an outgoing connection from an embedded device, it is in the form of a direct binary connection over TCP. The use of a direct binary connection improves the performance of message transfer. This binary information is converted to an XML format and subsequently forwarded over HTTP in compliance with the FIPA specifications. When receiving incoming messages for an embedded device from other platforms, this process is reversed, thus the HTTP is converted into a binary format. In this respect, the AFME message transport service supports transparent communication between agents operating on embedded devices and those operating on the standard version of the system or other FIPA compliant platforms. Agents are unaware of whether the cohort they are communicating with is behind a firewall or not. They correspond with the mailbox server in the same manner as if they were communicating with a platform directly.

4.3.2 Migration

Agent migration is often classified as either strong or weak. This classification is related to the amount of information transferred when an agent moves from one platform to another. Truly strong migration is not possible in Java. Within AFME support is only provided for the transfer of the agent's mental state (see Figure 2). Any classes required by the agent must already be present at the destination. This is because CLDC does not contain an API for introspection and is thus prevented from dynamically loading foreign objects. The reason CLDC applications do not contain security managers or class loaders that would enable the dynamic execution of foreign code is that the JVM specification does not contain a class verifier. The verifier forms one of the most important security aspects of the original J2SE JVM architecture. It ensures the integrity of a sequence of byte codes by performing a data-flow analysis on them at runtime. Conversely, within CLDC the code must be pre-verified in order to execute. This improves the performance of the system in that the code does not have to be continuously checked while executing. It prevents the system however from dynamically downloading and executing foreign objects because the system cannot verify that the objects' code is safe to use. Malicious developers could simply alter the structure of

a class, for example by changing the operand of a jump opcode such as goto, to crash the JVM and potentially the operating system of the mobile device. Thus no support for introspection is provided.

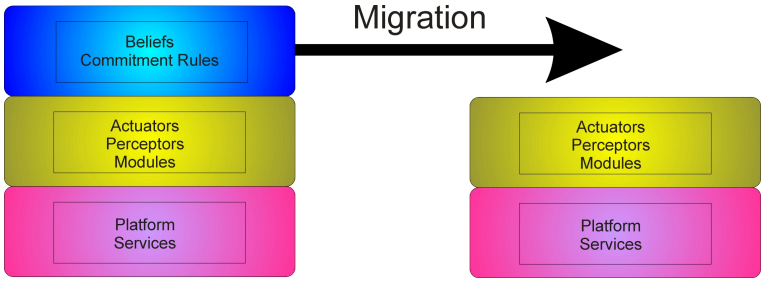

Fig. 2 AFME Migration

To facilitate the migration process within AFME a similar approach is taken to that of the message transport service. This is because agents must also be capable of penetrating the service provider firewall. Thus, agents first migrate to a migration server where they wait for a connection from their destination platform. When a connection is received they are transferred accordingly. When migrating back the agents also go through the migration server.

As agents move to and from embedded devices the BDI commitment rules that govern the agents' behaviour are altered to enable the agents to adapt to their environments. The agents' designs are decoupled into the core behaviours that operate on all platforms and platform specific behaviours that contain dependencies on a particular framework. The core behaviours represent the lowest common denominator or the essence of an agent's functionality that will execute on all devices. The platform specific behaviour represents the commitment rules that need to be altered when an agent moves from one type of environment to another. The essence of the agent is always contained within the agent design, whereas agents maintain beliefs about where the platform specific commitment rules may be obtained. These beliefs are in the form of URLs and represent the location at which the rules may be downloaded. These URLs are used within the mailbox server in the creation of agent designs that combine the essence of the agents with their platform specific functionality.

4.4 Standards compliance, interoperability and portability

Although the AFAPL is a general language, at the moment the interpreters and all of the tools for both Agent Factory and AFME are based on Java. Communication in the system is consistent with the FIPA specifications.

- At present, the platform requires Java to operate. The desktop version of the system will work with either standard Java or the JME CDC platform; it cannot work with CLDC. AFME has been specifically designed for the JME CLDC Java platform, but it can also work with standard Java.
- The message transport service and architecture of the system is consistent with the FIPA specifications. We discuss the message transport service in greater detail in the previous subsection. Additionally, support for yellow and white page services is provided.
- The System has been integrated with Java Servlets, JSP, and Struts. Future work will investigate the incorporation of the system with OSGi [2] for service management and deployment.
- Standard Java is required for the current framework and the Netbeans IDE is required for some of the tools to operate. In theory, an interpreter for AFAPL could be written in any imperative programming language. In the past, a Smalltalk-80 AFAPL interpreter was maintained, but at present, however, Java is necessary.
- The platform supports open multi-agent systems and heterogeneous agents.

4.5 Other features of the platform

Agent Factory is a modular open source platform. It supports the development and deployment of different types of agent architectures. Nevertheless, the platform is generally used for the development of intentional agents. The AFAPL interpreter is quite efficient when compared to other work in this area, but it is acknowledged that AFAPL does not conform to the same semantics as other systems. The footprint of the software has been reduced with the development of AFME. We discuss this in greater detail in Section 6.

- Agent Factory is a modular system and has been specifically designed to enable its core functionality to be extended with new features. It is open source and is freely available from SourceForge.
- The overhead of Agent Factory depends on the type of agents that are developed using the framework. AFAPL is quite practical and efficient when considered in the context of intentional agent platforms.
- The platform is currently a stable open source distribution.

- The structure of the agent community developed using Agent Factory is dependent on the design of the individual agents. As such, centralised, distributed, or hierarchical control can be catered for, but it does not provide libraries for interaction protocols or group templates.
- The platform enables the reuse of agent designs through the use of inheritance and dynamic role adoption.

5 Applications supported by the language and/or the platform

Over the years, the Agent Factory Framework has been used for the development of several research projects. The results and requirements of these projects have influenced the design of the system and have motivated the introduction of new features and functionality.

- At present, the majority of applications Agent Factory has been used for have been research projects that reflect the requirements of real world applications.
- Agent Factory does not target a specific domain and has been deployed in an array of diverse application domains. These have included E-Commerce [26], Ubiquitous Computing [36], Mobile Computing [38], Robotics [19], Wireless Sensor Networks [37], and Mixed Reality [17].

6 Comparison to Related Work

Agent Factory distinguishes itself from other intelligent agent frameworks [25, 32, 45, 1] in several ways. It is founded upon a logical formalism of belief and commitment [10] and enables agents to be programmed directly in terms of their commitments (see Section 3 for a discussion of the commitment management process). Additionally, it was one of the earliest frameworks to consider software engineering issues at its core, drawing from, and building upon, early work on Agent Oriented Software Engineering [41]. The framework comprises a four layer architecture that includes a development methodology, integrated development environment support, an agent-oriented programming language, and a runtime environment.

There currently a trend in the development of programming languages that use an XML syntax [7, 21, 48, 14, 47, 29, 16]. Although these systems have made many significant and innovative contributions to the field, the use of XML has to be called to account. This approach has not been adopted in Agent Factory/AFME. XML is useful as a mechanism for data exchange between computer programs, but not as a language to be used by humans. It's

a sophisticated data format nothing more (see Terence Parr's soapbox [42] for more details). The correct approach is to write a parser and lexical analyzer, lest pay a heavy price in productivity due to the cumbersome nature of XML. Even if XML were to be used, the data must still be interpreted. That is, even though XML has an extensible syntax, the developer must still write code to "make sense" of, or use, the information obtained from the parser when additional functionality or requirements are added. The semantics must still be encoded in the application or agent, regardless of the manner in which the agent design is represented. James Davidson, the creator of the well known Apache Ant, one of the most widely used XML-based tools, notes about the XML syntax of Ant in [15]:

> "If I knew then what I know now, I would have tried using a real scripting language, such as JavaScript via the Rhino component or Python via JPython, with bindings to Java objects that implemented the functionality expressed in today's tasks. Then, there would be a first-class way to express logic, and we wouldn't be stuck with XML as a format that is too bulky for the way that people really want to use the tool."

XML is overly verbose and is far from ideal as the syntax of a programming language, specification language, or for configuration files to be used by people. As noted by Parr [42]:

> "Humans have an innate ability to apply structure to a stream of characters (sentences), therefore, adding mark-up symbols can only make it harder for us to read and more laborious to type. The problem is that most programmers have very little experience designing and parsing computer languages. Rather than spending the time to design and parse a human-friendly language, programmers are using the fastest path to providing a specification language and implementation: "Oh, use XML. Done." And that's OK, but I want programmers to recognize that they are providing an inferior interface when they take that easy route."

Several agent frameworks have been developed for the fabrication of agents for desktop environments. With the explosive growth of the mobile phone market, there has been a drive to develop similar frameworks for constrained environments and in particular JME CLDC. Typically, these systems have two versions of the agent platform, one for the desktop environment and one for the CLDC environment. This is the case with Agent Factory/AFME. Other examples include Jade/Jade LEAP [4], 3APL/3APL-M [28], SAGE/SAGE-Lite [27], and Cougaar/CourgaarME [49]. Agent Factory/AFME distinguishes themselves from these other frameworks in a number of ways. JADE-LEAP, CougaarME, MicroFIPA-OS, and SAGE-Lite are frameworks for the development of agent technology, but they are not reflective and do not use an abstract agent programming language that is based on a theory of rational agency[1]. 3APL-M is similar to AFME/Agent Factory

[1] Intelligent extensions built for JADE will not work with the CLDC version of LEAP without making modifications to the code due to the different APIs supported by standard Java and CLDC.

in that it does contain reasoning capabilities, but it does not contain a networking component. If the developer wishes an agent to communicate over the network, they must write the code from scratch. Agent Factory/AFME provides support for networking through the message transport service.

At present, most of these frameworks target mobile phones and PDAs. Agilla [20] is an agent platform that has been developed for WSNs, but it also does not contain reasoning capabilities and therefore does not conform to the same definition of agency as Agent Factory/AFME.

The design of AFME has been strongly influenced by the 'Law of Demeter' [30], which specifies the coding guideline "only talk to your immediate friends". The 'Law of Demeter', or Principle of Least Knowledge, leads to the creation of loosely coupled classes that improve the maintainability of the software [6, 43]. Using the Law as a general principle, tends to favour composition over inheritance and avoids the use of accessor (get/set) methods[2]. Rather than a callee obtaining data from an object to perform some operation through the use of an accessor method, the callee directs the object to perform the operation on its behalf [24]. That is, objects are designed in terms of their capabilities and their state is not exposed through the use of accessors. This is often referred to as 'delegation' by object-oriented developers. It leads to a more declarative approach to object development and also tends to reduce the footprint of the software by minimising code duplication.

The footprint of AFME is quite low. For instance, the core infrastructure has a Jar size of 77k, an NCSS value [22] of 2601, and a McCabe cyclomatic complexity [31] (not to be confused with algorithmic complexity) of 2.91. If just the core reasoning capabilities of the platform are considered and the Jar file is obfuscated, the Jar size can be reduced to 17k. When considering the footprint, we must also take into account the overhead and resource requirements of the software necessary to run the platform. With AFME, the JME CLDC Java platform is required. CLDC is considerably less resource intensive than standard Java. As noted earlier, there have been a number of platforms developed for CLDC. 3APL-M is the closest framework to AFME in that it is based on a theory for rational agency and contains reasoning capabilities. Through our experience and experimentation [33], it has been found that AFME is considerably faster than 3APL-M in terms of execution time. Nevertheless, it is acknowledged that 3APL-M contains features not supported by AFME. To the best of our knowledge, AFME is the smallest footprint reflective agent-oriented platform. Nevertheless, it is accepted that AFME does not conform to the same semantics as other reflective platforms. For instance, 3APL-M has been incorporated with a complete Prolog engine, AFME has not.

[2] It should be noted that the LoD does not only apply to accessors (see [30] for more details).

7 Conclusion

Agent Factory is a flexible FIPA compliant open source Java-based platform for the development, visualisation, and deployment of multi-agent systems. It is an evolving project and over the years has gone through a metamorphosis process, producing sibling frameworks such as AFME. Agent Factory has been designed with Software Engineering principles at its core and represents a modular framework that supports the development of a diverse variety of agent-oriented applications. The footprint of the system has been further reduced with the development of AFME. This chapter discussed the various modifications and enhancements that were made to the framework together with the motivations for such. As new technologies emerge, it is envisaged that Agent Factory will be further augmented to reflect new threads and developments. In the short term, we are investigating the potential to further enhance and optimise performance for embedded and resource constrained devices. In the longer term, as the number of heterogeneous devices and platforms increases, we are looking at the potential of using technologies, such as OSGi [2], for service management and deployment.

Acknowledgements The authors gratefully acknowledge the kind support of Science Foundation Ireland (SFI) under grant no. 07/CE/I1147 and the Irish Research Council for Science, Engineering, and Technology (IRCSET).

Appendix (Language Summary)

1(a) The language supports mental attitudes and specifically beliefs and commitments. Reactive behaviour is encoded imperatively in Java.

1(b) The language supports speech act based primitives for communication. Wildcard pattern matching is used to broadcast messages to multiple agents, for example a message sent to Ja* would be received by Jack and Jay, but not Frank.

1(c) Support is provided for the construction of mobile agents. Truly strong migration is not possible in Java. With Agent Factory, the agents' mental state and code are transferred. With AFME, only the mental state is transferred.

1(d) The language is easy to understand for someone who has experience of declarative/logic programming.

1(e) The language has clear and precise semantics. The original language was formalised in Collier's thesis [10]. Work is underway on formalising more recent enhancements/alterations.

1(f) The language is intended for the construction of a variety of agent programs.

1(g) The language enables the reuse of agent designs through roles, but does not enable the definition of new language components.
1(h) At present a clear path to formal verification is not provided.
1(i) Software Engineering principles have been considered in the design of the language.
1(j).i The language is integrated with Java and was previously integrated with Smalltalk-80.
1(j).ii The language allows the invocation of methods written in classical languages.
2(a).i Detailed instructions on how to install and use the framework (including Javadoc) are available from http://www.agentfactory.com.
2(a).ii In its current form, the framework requires either Java Standard Edition or Java Micro Edition to execute.
2(b) The general architecture of the framework is consistent with the FIPA specifications. Agent Factory provides FIPA compliant communication through a message transport service. With AFME, FIPA communication is facilitated through a combination of a TCP (binary) message transport service and a "translator message server" that operates on a desktop machine. Support is also provided for yellow and white page services.
2(c) The platform can be extended with additional functionality through open source collaboration.
2(d).i An agent mental state debugger is provided along with a logger.
2(d).ii Extensive documentation including Javadoc is available from agentfactory.com.
2(d).iii A Netbeans IDE plugin is provided.
2(e) Existing applications have been integrated with JSP, Java Servlets, and Struts.
2(f) An application running the platform would require Java.
2(g).i Agent Factory is quite an efficient and practical system and is capable of executing a large number of agents subject to the number of commitment rules the agents have and the hardware on which they are operating.
2(g).ii The platform is a stable open source distribution.
2(h).i The platform supports open multi-agent systems and heterogeneous agents.
2(h).ii The structure of the agent community is dependent on the design of the individual agents that form the community. As such, centralised, hierarchical, or distributed control can be catered for.
2(h).iii The platform does not provide libraries of interaction protocols. Agent templates are supported and reuse is facilitated through the use of roles.
3(a) The platform has been used for the development of applications in several disparate domains, including E-Commerce [26], Ubiquitous Computing [36], Mobile Computing [38], Robotics [19], Wireless Sensor Networks [37], and Mixed Reality [17].

3(b) The platform does not target a specific domain.

References

1. Albuquerque, R., Hbner, F., de Paula, G., Sichman, J., Ramalho, G.: Ksaci: A handheld device infrastructure for agents communicaiton. Pre-proceedings of the Workshop on Agent Theories, Architectures, and Languages, (ATAL) (2001)
2. Alliance, O.: OSGi Service Platform, Release 3. IOS Press, Inc. (2003)
3. Bellman, R.E.: Some Vistas of Modern Mathematics. University of Kentucky Press (1968)
4. Berger, M., Rusitschka, S., Toropov, D., Watzke, M., Schlichte, M.: The Development of the Lightweight Extensible Agent Platform. EXP in Search of Innovation **3**(3), 32–41 (2003)
5. Birkhoff, G., Lipson, J.: Heterogeneous algebras. Journal of Combinatorial Theory **8**(1), 15–133 (1970)
6. Booch, G.: Object-oriented Analysis and Design, 2nd edition. Addison Wesley (1994)
7. Braubach, L., Pokahr, A., Lamersdorf, W.: Jadex: A BDI agent system combining middleware and reasoning. In: M.K. R. Unland M. Calisti (ed.) Software Agent-Based Applications, Platforms and Development Kits, pp. 143–168. Birkhuser-Verlag, Basel-Boston-Berlin (2005). Book chapter
8. Collier, R., Ross, R., O Hare, G.M.P.: A role-based approach to reuse in agent-oriented programming. AAAI Fall Symposium on Roles, an interdisciplinary perspective (Roles 2005) (2005)
9. Collier, R.W.: The realisation of Agent Factory: An environment for the rapid prototyping of intelligent agents. M. Phil., Univ. of Manchester (1996)
10. Collier, R.W.: Agent Factory: A Framework for the Engineering of Agent-Oriented Applications. Ph.D. Thesis (2001)
11. Collier, R.W., O Hare, G.M.P.: Agent Factory: A Revised Agent Prototyping Environment. In: 10th Irish Conference on Artificial Intelligence & Cognitive Science, pp. 1–3 (1999)
12. Collier, R.W., O Hare, G.M.P., Lowen, T., Rooney, C.: Beyond prototyping in the factory of the agents. 3rd Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'03), Lecture Notes in Computer Science (LNCS) **2691** (2003)
13. Collier, R.W., Rooney, C.F.B., Donoghue, R.P.S., O'Hare, G.M.P.: Mobile BDI agents. In Proceedings of the 11th Irish Conference on Artificial Intelligence and Cognitive Science (2000). URL `citeseer.ist.psu.edu/collier00mobile.html`
14. Cossentino, M., Burrafato, P., Lombardo, S., Sabatucci, L.: Introducing Pattern Reuse in the Design of Multi-agent Systems. Lecture Notes in Computer Science pp. 107–120 (2003)
15. Davidson, J.D.: The Creator of Ant Exorcizes One of His Demons. Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Apps by M. Clark. The Pragmatic Programmers (2004)
16. De Meo, P., Garro, A., Terracina, G., Ursino, D.: X-Learn: An XML-Based, Multi-agent System for Supporting "User-Device" Adaptive E-learning. Lecture Notes in Computer Science pp. 739–756 (2003)
17. Dragone, M., Holz, T., Duffy, B.R., O Hare, G.M.P.: Ubiquitous Realities through Situated Social Agents. In: Proc. of the 5th Intl. Working Conference of Computer Animation and Social Agents (CASA05) (2005)

18. Dragone, M., Lillis, D., Collier, R.W., O Hare, G.M.P.: Sosaa: A framework for integrating agents and components. In: 24th Annual Symposium on Applied Computing (ACM SAC 2009), Special Track on Agent-Oriented Programming, Systems, Languages, and Applications. Honolulu, Hawaii, USA (2009)
19. Duffy, B.R., Collier, R.W., O Hare, G.M.P., Rooney, C.F.B., O'Donoghue, R.P.S.: Social Robotics: Reality and Virtuality in Agent-Based Robotics. In: Bar-Ilan Symposium on the Foundations of Artificial Intelligence: Bridging Theory and Practice (BISFAI) (1999)
20. Fok, C.L., Roman, G.C., Lu, C.: Rapid development and flexible deployment of adaptive wireless sensor network applications. In: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'05), pp. 653–662. IEEE (2005)
21. Garro, A., Palopoli, L.: An XML Multi-agent System for E-learning and Skill Management. Lecture Notes in Computer Science pp. 283–294 (2003)
22. Grady, R.B.: Successfully applying software metrics. Computer **27**(9), 18–25 (1994). DOI http://dx.doi.org/10.1109/2.312034
23. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM **21**(8), 666–677 (1978)
24. Holub, A.: Holub on Patterns: Learning Design Patterns by Looking at Code. APress (2004)
25. Howden, N., Ronnquist, R., Hodgson, A., Lucas, A.: JACK Intelligent Agents-Summary of an Agent Infrastructure. In: 5th International Conference on Autonomous Agents (2001)
26. Keegan, S., O'Hare, G.M.P., O'Grady, M.J.: Easishop: Ambient intelligence assists everyday shopping. Information Sciences **178**(3), 588–611 (2008)
27. Khalique, S., Farooq, S., Ahmad, H.F., Suguri, H., Ali, A.: Sage-lite: An architecture and implementation of light weight multiagent system. ISADS **0**, 239–244 (2007). DOI http://doi.ieeecomputersociety.org/10.1109/ISADS.2007.68
28. Koch, F., Meyer, J.J., Dignum, F., Rahwan, I.: Programming Deliberative Agents for Mobile Services: the 3APL-M Platform. AAMAS'05 Workshop on Programming Multi-Agent Systems (ProMAS05) (2005)
29. Konnerth, T., Endert, H., Heler, A.: JIAC IV Programmers guide (2007)
30. Lieberherr, K., Holland, I., Riel, A.: Object-oriented programming: An objective sense of style. in Object Oriented Programming Systems, Languages and Applications Conference, in special issue of SIGPLAN notices pp. 323–334 (1988)
31. McCabe, T.J.: A software complexity measure. IEEE Transactions on Software Engineering **2**(4), 308–320 (1976)
32. Morley, D., Myers, K.: The spark agent framework. In: AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 714–721. IEEE Computer Society, Washington, DC, USA (2004). DOI http://dx.doi.org/10.1109/AAMAS.2004.267
33. Muldoon, C.: An Agent Framework for Ubiquitous Services. Ph.D. thesis, School of Computer Science and Informatics, Dublin, Ireland (2007)
34. Muldoon, C., O Hare, G.M.P., Bradley, J.F.: Towards Reflective Mobile Agents for Resource Constrained Mobile Devices. In: AAMAS 07: Proceedings of the Sixth International Joint conference on Autonomous Agents and Multiagent Systems. ACM, Honolulu, Hawai'i (2007)
35. Muldoon, C., O Hare, G.M.P., Collier, R.W., O Grady, M.J.: Agent Factory Micro Edition: A Framework for Ambient Applications. In: Intelligent Agents in Computing Systems, *Lecture Notes in Computer Science*, vol. 3993, pp. 727–734. Springer, Reading, UK (2006)
36. Muldoon, C., O Hare, G.M.P., Phelan, D., Strahan, R., Collier, R.W.: ACCESS: An Agent Architecture for Ubiquitous Service Delivery. In: Proceedings Seventh International Workshop on Cooperative Information Agents (CIA), *Lecture Notes in Computer Science*, vol. 2782, pp. 1–15. Springer, Helsinki (2003)

37. Muldoon, C., Tynan, R., O Grady, M.J., O Hare, G.M.P.: Realising an agent-oriented middleware for heterogeneous sensor networks. In: ACM/IFIP/USENIX 9th International Middleware Conference, pp. 82–83. ACM Press (2008)
38. O Grady, M.J., O Hare, G.M.P.: Gulliver's genie: agency, mobility, adaptivity. Computers & Graphics **28**(5), 677–689 (2004)
39. O Hare, G.M.P.: Agent Factory: An Environment for the Fabrication of Distributed Artificial Systems. O Hare, Gregory M. P. and Jennings, N. R.(Eds.), Foundations of Distributed Artificial Intelligence, Sixth Generation Computer Series, Wiley Interscience Pubs pp. 449–484 (1996)
40. O Hare, G.M.P., Abbas, S.: Agent Oriented Programming: Communicating Intentional Processes (1994)
41. O Hare, G.M.P., Wooldridge, M.J.: A software engineering perspective on multi-agent system design: experience in the development of MADE. Kluwer Computer And Information Science Series pp. 109–127 (1992)
42. Parr, T.: Soapbox: Humans should not have to grok XML. `http://www.ibm.com/developerworks/xml/library/x-sbxml.html` (2001)
43. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object Oriented Modeling and Design. Prentice Hall (1991)
44. Shoham, Y.: AGENT0: A simple agent language and its interpreter. In: Proceedings of the Ninth National Conference on Artificial Intelligence, vol. 2, pp. 704–709 (1991)
45. Sierhuis, M.: Brahms: A Multi-Agent Modeling and Simulation Language for Work System Analysis and Design. Ph.D. thesis (2001)
46. Teacy, W., Farinelli, A., Grabham, N., Padhy, P., Rogers, A., Jennings, N.: Max-sum decentralised coordination for sensor systems. In: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: demo papers, pp. 1697–1698. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC (2008)
47. Tuguldur, E., Patzlaff, M.: Collecting Gold: MicroJIAC Agents in Multi-Agent Programming Contest. Lecture Notes in Computer Science **4908**, 251 (2008)
48. Weiliang, M., Sheng, H.: An XML-Based Language for Coordination Protocol Description in Multi-agent System. In: Proceedings of the 14th International conference on Industrial and engineering applications of artificial intelligence and expert systems: engineering of intelligent systems, pp. 708–717. Springer-Verlag London, UK (2001)
49. Wright, W., Moore, D.: Design considerations for multiagent systems on very small platforms. In: AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems, pp. 1160–1161. ACM Press, New York, NY, USA (2003). DOI http://doi.acm.org/10.1145/860575.860845

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the definition; numbers in roman refer to the pages where the entry is used.