Beyond Prototyping in the Factory of Agents

Rem Collier
University College Dublin
Belfield, Dublin 4,
+353 1 716 2491
rem.collier@ucd.ie

Gregory O'Hare
University College Dublin
Belfield, Dublin 4,
+353 1 716 2472
gregory.ohare@ucd.ie

Terry Lowen
University College Dublin
Belfield, Dublin 4,
+353 1 716 2491
terry.lowen@ucd.ie

Colm Rooney
University College Dublin
Belfield, Dublin 4,
+353 1 716 2854
colm.rooney@ucd.ie

ABSTRACT

This paper introduces Agent Factory, a cohesive framework supporting a structured approach to the development and deployment of agent-oriented applications. We describe Agent Factory together with an accompanying agent development methodology. We detail the key attributes of Agent Factory, namely: visual design, design reuse, behaviour enactment, migration, and ubiquity. Agent Factory functionality is exercised by way of a case study. We offer cross comparison of our system with exemplar agent prototyping environments.

Categories and Subject Descriptors

1.2.11 [**Distributed Artificial Intelligence**]: Intelligent Agents, and Multi-Agent Systems.

General Terms

Design.

Keywords

Agent Architectures, Agent-Based Software Engineering, Mobile Agents, Agent Languages and Environments, Lightweight Agents.

1. INTRODUCTION

The provision of cohesive support for the construction of multi-agent systems is essential if agent technologies are to be employed within industry. To this end, it is vital that software engineering frameworks be devised that promote structured approaches to the development and deployment of agent-oriented applications. This paper introduces one such framework, the *Agent Factory System* [5] [18].

In section 2 we provide an overview of the Agent Factory framework. Subsequently, section 3, describes the *Agent Fabrication Process,* which underpins structured agent development. Sections 4 and 5 respectively describe the Development and Run-Time Environments. We conclude with a brief case study, and related research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Conference '00, Month 1-2, 2000, City, State.

2. THE FRAMEWORK

Agent Factory provides a cohesive framework for the development and deployment of agent-oriented applications. Specifically, it delivers extensive support for the creation of Belief-Desire-Intention (BDI) agents. This type of agent employs the Intentional Stance [10] from an internal perspective through the explicit modelling of a mental state based upon various mental attitudes [5].

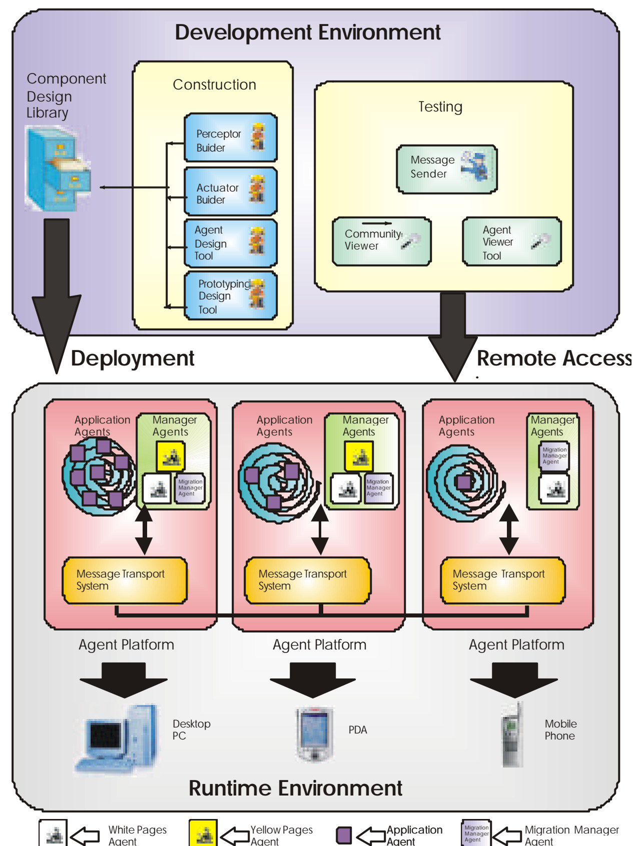

Figure 1: The Agent Factory Framework

This framework was originally implemented in Smalltalk-80, however, in order to support the deployment of agents on computationally challenged devices such as Personal Digital Assistants (PDAs), or Micro Robots (e.g. Khepera [14]), the

Run-Time Environment has been recast in Java. A diagrammatic overview of Agent Factory is presented in figure 1.

As is illustrate in this diagram, Agent Factory is organised into two core environments: the Agent Factory Development Environment, and the Agent Factory Run-Time Environment. The former environment delivers a set of Computer-Aided Software Engineering (CASE) tools that support the Agent Fabrication Process, whilst the latter delivers support for the deployment of agent-oriented applications over a wide range of network-enabled Java-compliant devices.

3. THE AGENT FABRICATION PROCESS

The Agent Factory system constitutes a membrane within which several distinct layers exist. The innermost layers provide the necessary deductive apparatus to execute BDI agents and an Agent Communication Language (ACL) to facilitate inter-agent communication.. Above and enveloping this is *the Agent Factory Development Environment* a cohesive toolset that assist in the various stages of design, implementation and deployment. The outermost layer provides an associated methodology, the *Agent Fabrication Process (AFP),* which imposes a partial ordering upon the invokation of such tools and consequently oversees a logical and disciplined sequence of steps that should be followed when implementing an agent-oriented application with Agent Factory.

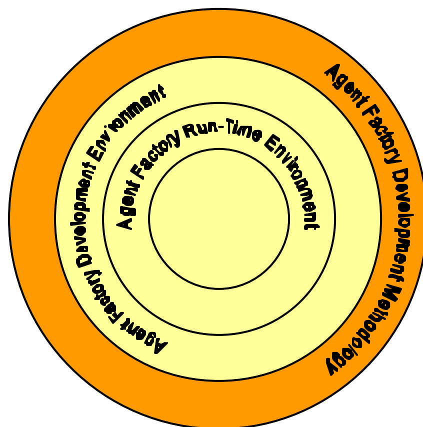

Figure 2: A Layered View of the Agent Factory System

Much recent work has been undertaken in the area of agent methodologies including: Agent UML [1], GAIA [23] Prometheus [20], ROADMAP [13] and MaSE [9]. The Agent Fabrication Process is complete in the sense that it does offer end-to-end support for the development process and in this respect is similar to Prometheus. However the Agent Fabrication Process is closely coupled with the Agent Factory Development Environment and unlike many agent methodologies that are largely paper based we offer software tool support for all development stages but analysis. At present we use Behaviour diagrams a diagramming convention that is currently without tool support.

The AFP is comprised of 5 broad steps namely:

- *Ontology Definition.*
- *Actuator and Perceptor Building.*
- *Agent Coding.*
- *Agent Testing.*
- *Application Deployment.*

Figure 3 depicts these stages and specifically the tool support provided for each stage. Within the context of this paper it is not appropriate to attempt to describe all of this rich toolset, rather we focus upon the support for agent coding, testing and deployment.

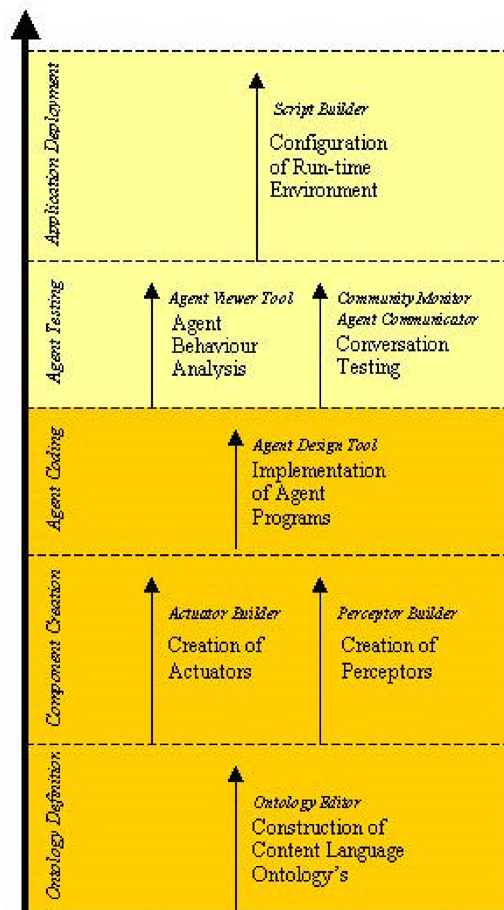

Figure 3: The Agent Fabrication Process (AFP)

4. DEVELOPMENT ENVIRONMENT

Tool-based support for all steps of the Agent Fabrication Process (AFP) is delivered through an integrated toolset known as the Agent Factory (AF) Development Environment. This environment is separate to the Run-Time Environment (see figure

1), and focuses upon the development of various components and their composition into coherent *agent designs*.

As stated in section 2, Agent Factory provides support for the fabrication of BDI-agents. This agent type is realised through the implementation of some mental state architecture and a corresponding agent interpreter that manipulates the mental state, allowing the agent to reason about how best to act. The mental state itself is modelled using multi-modal temporal logic. Within AF, this logic is realised as a set of programming constructs within a purpose-built agent programming language. Details of this programming language can be found in [5].

From an agent construction perspective, agents are viewed as instances of agent designs. These designs are a combination of an *agent program,* written in the above agent programming language, and a set of actuators and perceptors. Actuators represent the primitive actions that are directly executable by an agent. Perceptors deliver the mechanism by which raw sensory data is transformed and encoded in the agent's mental state.

Currently Agent Factory provides tools that support: the definition of ontology's, the development of actuator and preceptor units, the coding of agent programs, and the testing and deployment of published agent designs.

The components that underpin an agent design (namely, the actuators, and perceptors), and the designs themselves are stored within a specific module of the Development Environment known as the *Component Design Library (CDL)*. Agent designs that are published to a CDL may, at some later point in time, be deployed into the Run-Time Environment using the *Deployment Tool*. Currently, the CDL is realised as a set of folders that are stored within an instance of the Concurrent Versions System (CVS) [7], a client-server version-control system.

In the following subsections, we describe in more detail the key features that distinguish the Agent Factory Development Environment from other Agent Prototyping Environments: design reuse, behaviour enactment, and visual design.

4.1 Reuse in Agent Factory

An agent is realised as an instance of an *agent design*. Many agent designs have common functionality, for example, support for agent communication is delivered through a set of actuators (one for each communicative act), and through a perceptor that monitors the agents message queue. Consequently, any agent design that requires inter-agent communication must include these components.

This motivates the inclusion, of support for the reuse of agent designs through *inheritance*. To this end, an agent design is decomposed into a set of *agent classes* that are organised into a *class hierarchy*. Each agent class specifies a set of components and a partial agent program. This class is basically a templated solution that may be extended as required in the subclasses. An example class hierarchy can be found on the left-hand side of figure 4. The right-hand side of the tool presents the developer with a structured set of views that allows them to edit various facets of the agent design (i.e. to add/remove actuator and perceptor components or to modify the agent program).

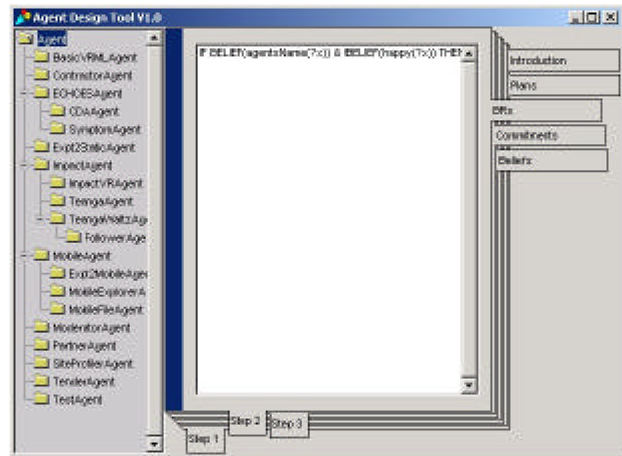

Figure 4: Screenshot of the Agent Design Tool

To instantiate an agent from a design, the developer must first publish the design. This involves the compilation of the agent design from its constituent agent classes. Thus, the compiled agent design is a union of all the partial agent programs and components specified within the relevant agent classes.

Each agent class specifies a set of components and a partial agent program. From this perspective, an agent design is viewed as a *path* within the class hierarchy. The construction of agent classes from prefabricated components is central to the Agent Fabrication Process presented in section 3. This forces an ordering on the use of the tools provided within the Development Environment where a component must be built before it may be associated with an agent class.

A drawback of inheritance arises from cases where two agent designs use the same code, but do not share the same parent class in the class hierarchy. For example, it is possible for a diverse range of agents to make use of some code that relates to the provision of services (i.e. code relating to the handling of requests for some service). Similar issues have commonly arisen in Object-Oriented Programming [22]. One accepted solution is not to use inheritance, but instead to use *composition*.

From this perspective, an agent design can be viewed as an aggregation of pre-existing modules and protocols that are joined through *glue code*. Initial work carried out on this revised approach to the fabrication of agents is discussed later in section 4.3.

4.2 Visualising Behaviour Enactment

A second key feature of Agent Factory is the tool support that has been provided for the visualisation of agents. Within the context of the Agent Fabrication Process (section 3), these deliver support for the testing phase. Agent Factory provides tools that allow the developer to explore and monitor how the agents will

react to various potential scenarios. Currently, three tools have been developed to monitor the enactment of agent behaviours, namely: the Agent Viewer Tool, the Community Monitor, and the Message Sender.

The Agent Viewer Tool is used to facilitate the visualisation, configuration, and execution of agents (see figure 5). It consists of a set of interfaces and controls. The controls empower the user to start, step, and pause the agent as well as influence it's mental state. The interfaces are delivered via the adoption of a notebook metaphor. Each tab on the notebook corresponds to a particular view of the agent, such as its mental state and any relevant agent subsystems (such as the message queue). For example, the interface in figure 5 allows the developer to view the beliefs of the agent. This tool, and indeed the metaphor, support *information hiding* and allow the designer to view only those aspects that they are interested in at a given instance. Furthermore it provides an *abstraction* mechanism where designers are presented with the mental state in a more palatable form.

The Community Monitor enables the developer to monitor interaction between specified sets of agents. This is currently realised in form of a view on time-stamped transcripts of the messages received by each of the selected agents.

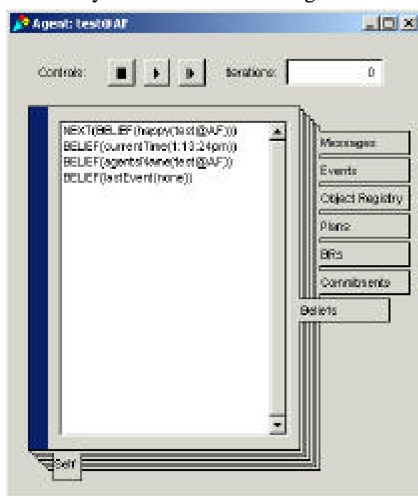

Figure 5: Screenshot of the Agent Viewer Tool

In addition to providing visual tools that allow the user to examine and interact with existing communities of agents, Agent Factory also provides visual agent programming tools that assist the user in creating agent designs that may be published and deployed. Examples of such tools are described in section 4.3 and these go beyond the tools described in section 4.1.

4.3 Visual Tools for Agent Design

Agent interactions play a crucial role in many Multi-agent Systems. As a result, a range of graphical formalisms have been adopted that allow agent developers to abstract away from implementation details and focus on the core aspects of such interactions, e.g. Petri Nets [8], Timed Petri Nets [6], Push Down Transducer [16], Agent UML (AUML) [1]. In order to facilitate the fabrication and deployment of collaborative agent communities, Agent Factory provides a tool that assists agent developers in the creation of agent interaction protocols through the application of such graphical formalisms.

The *Agent Factory Protocol Tool* is a graphical tool that allows users to diagrammatically construct agent interaction protocols, which are then compiled into usable agent designs. Three components provide the core of the protocol editing functionality: The *Interface Component* acts as the medium through which the user views and edits the protocol. The *Model Component* stores the elements that make up a protocol diagram (e.g. messages, threads) and the associations between them. The *Semantic Checker* maintains the *model* and its consistency, in the sense that it enforces the implementation of the protocol meta-model (i.e. the rules that constrain elements and associations within a protocol diagram, e.g. the AUML interaction protocol specification). The flow of control is as follows: Any non-cosmetic user updates to the protocol (i.e. those that affect the *model*) are forwarded from the Interface to the Semantic Checker. The Semantic Checker then queries the current state of the *model* and evaluates the user's request in light of this information and the constraints set down in the meta-model. If the request is successful the *model* is updated accordingly and such updates are communicated to the Interface component, which displays the result to the user. In the event of a rejected request the Interface is also notified and can inform the user of such.

The Protocol Tool has been designed and implemented in Java using a modular event-based architecture. The loose coupling between the system components facilitates a *plug-and-play* approach. This entails that we are not constrained to a single implementation of the Interface, *model*, or the Semantic Checker. We could, for example, alter how the user interacts with the protocol without changing the underlying protocol representation, or change the underlying protocol representation without changing how the user interacts with the protocol. This permits a lot of flexibility in the types of graphical formalism applied.

Within this architecture, the components themselves have been implemented in such a fashion as to provide as much flexibility as possible, so that different graphical formalisms may be applied without needing to alter or replace the components. The current Interface utilises an XML initialisation file that defines what protocol elements to include their associated details, e.g. display classes, events generated, menu icons. This initialisation file has a corresponding Constraints Definition File (CDF) that defines the meta-model. The Model component accepts any {Component, Component, Association} 3-tuple and stores their details in key-value pairs (Note, cosmetic details, e.g. x- and y-coordinates, are considered extraneous to the model and are maintained by the Interface). The Semantic Checker is implemented as an Agent Factory agent. This agent evaluates user requests based on the constraints in the CDF, the current state of the *model* and the rules set down in the agent design. Changing the agent rules can

alter the tool functionality without the need for recoding and without degrading the *model*, e.g. the commitment rules could determine which error message to send back to the Interface, or could apply additional application specific constraints to the protocol that are not sufficiently generic to include in the basic meta-model. Other facilities provided include a parser component for the input and output of protocol diagrams, image rendering and agent management (see figure 6).

Currently, the Protocol Tool is configured to allow users to define agent interaction protocols as per the AUML specification. Indeed, the flexible design of the system was inspired in part by the nascent state of this specification, so that new updates could be readily incorporated. In figure 6, we can see the Interface Component of the AF Protocol Tool together with the mental state of the Semantic Checker at a given instance.

While a tool for the graphical design of protocols is useful as a visual aid, what the developer really needs in the end is agent code. To this end the Protocol Tool provides a second service. When a protocol has been published (i.e. a version is ready to be applied) this published protocol can be compiled into an *agent skeleton*. This skeleton defines the external behaviour of an agent in terms of agent rules but leaves the (application specific) internal behaviour for the developer to implement (somewhat akin to an OOP interface). To aid in the implementation process, the Protocol Tool has a Rule Editor that allows users to associate code with diagram elements that represent blocks of agent deliberation. This code consists of user-defined modules linked together by pre- and post-conditions in order to maintain the coherence of the protocol. Currently, the Protocol Tool provides a mapping between AUML diagram elements and Agent Factory rule blocks. However, the Rule Editor interface and the model used to represent agent rule blocks are independent of this mapping. As such protocols developed using a different graphical formalism may be compiled into agent designs once a suitable mapping is specified.

5. RUN-TIME ENVIRONMENT

The term Run-Time Environment encompasses the architecture that delivers support for the deployment of agent-oriented applications over a wide range of Java compliant desktop, PDA and mobile telephone devices. It can be subdivided into a series of Agent Platforms that reside on each of the aforementioned devices (see figure 1).

5.1 The Agent Factory Agent Platform

The Agent Platform (AP) provides the physical infrastructure in which agents can be deployed. Each Agent Factory AP contains a Message Transport System, and three components implemented as Agent Factory agents, namely, a White Pages Agent, a Migration Manager Agent and may contain a Yellow Pages Agent.

The Message Transport Service is responsible for the delivery of messages between agents within the platform and to agents resident on other APs.

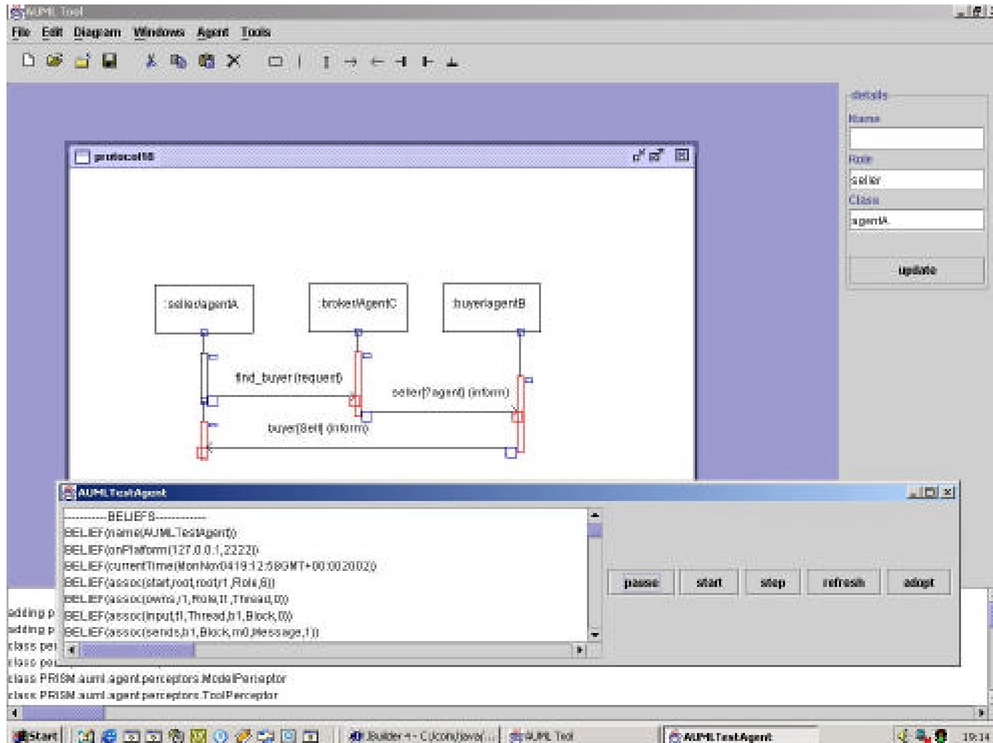

Figure 6: Screenshot of the Protocol Tool

The White Pages Agent (WPA) [2] [12] exerts supervisory control over access to and use of the Agent Platform. It is responsible for the creation and deletion of agents on the AP. It also maintains a directory of Agent Identifiers, which contain transport addresses for agents registered with the Agent Platform. The Migration Manager Agent (MMA) is responsible for overseeing the migration of agents to and from the AP. Together the WPA and the MMA fulfil the role of the Agent Management System in the Foundation for Intelligent Physical Agents (FIPA) specification [11].

The Yellow Pages Agent (YPA) as the name suggests provides a yellow pages directory service to agents. Agents may register their services with the YPA or query the YPA to find out what services are offered by other agents. The YPA fulfils the role of a Directory Facilitator in the FIPA specification.

5.2 Ubiquity and Agent Factory

The functionality of an Agent Factory AP takes cognisance of the computational power and memory restrictions of the device it is operating upon. A device such as a desktop PC has sufficient resources to support the full functionality of an AP with a sizeable community of agents. A PDA is more restricted in its resources but can still support the full functionality of an AP. It supports a significantly lesser population of agents than that of a desktop AP. A mobile phone is computationally more restrictied than a PDA and as such an AP residing on it does not offer the same level of functionality as the PDA or desktop configuration. Consequently, there is no YPA on the mobile phone AP and the size of the agent community is restricted to one or two agents. APs residing on mobile phones would very rarely be asked to provide a service for other agents, therefore a YPA on a mobile phone would prove redundant. If the mobile needs a YPA it can register to a YPA resident on a desktop AP.

5.3 Supporting Agent Migration

AF supports weak migration in that only the mental state and design of the agent should have to migrate. Agent migration from one AF Agent Platform to another is achieved through cloning. When an agent commits to migrate it informs the Migration Manager Agent (MMA) on the destination platform that it wishes to do so. The agent then sends its agent design to the destination MMA. Along with the mental state, the agent design contains the names of the actuators and perceptors, which the agent utilises during its execution. The destination MMA on receipt of the agent design checks to see if all the necessary actuators and perceptors for the agent are present on the destination Agent Platform. Omissions can be downloaded from the source of the migration. Once the destination platform has all the required functionality a new agent is then created using the downloaded design and mental state. The MMA then informs the source Agent Platform to dispose of the old agent and the new agent begins execution. Mobility in Agent Factory does not currently comply with the FIPA Agent Management Support for Mobility Specification in that an AF agent has no notion of a home platform and as such does not update its source platform to its new location. However, work is ongoing to ensure compliance in the near future.

6. CASE STUDY

In order to demonstrate and exercise the Agent Factory System a mobile computing demonstration was designed and trailed. The Where Are You (WAY) system [19][15] is a simple, yet effective application developed at University College Dublin for assisting mobile users in the location, tracking and rendezvousing with a variety of moving entities. In particular one of the most common and costly usage of mobile devices is the synchronisation and rendezvous of people. The WAY system seeks to provide an alternate solution to this problem by deploying mobile agent based technologies, namely Agent Factory. The WAY system enables location postings and updates to be passed between users with the minimum of fuss. These are subsequently depicted on a map-based interface (see figure 7). Additional functionality allows users to rendezvous with other entities (e.g. Public Transport).

Figure 7: A Screenshot of the WAY System

The WAY system is designed for mobile PDA deployment. In establishing a WAY service and a subsequent user connection, users firstly subscribe to the WAY system by contacting an Agent Factory Agent Platform on a server in wireless mode. Thereafter the subscriber receives a default WAY agent mental state that would migrate to the PDA. In addition an address book is supplied containing the Agent Identifiers of other users. The next step involves the issuing or acceptance of WAY requests from fellow users. Acceptance would enable the other user to know and track your whereabouts. To attain the same functionality they need to accept your WAY request. Upon disconnection the WAY agent migrates to the server to be stored persistently.

User trials have been conducted using a Compaq Ipaq 3760 equipped with a dual PCMCIA sleeve in order to accommodate a PCMCIA GPS and a Nokia card phone 2.0. The former provides the localisation data and the later the wireless communication infrastructure. Development on this device utilised the Jeode EVM.

At present detailed field trials are underway using the Dublin city centre as the test location and the results thus far are favourable. This application has verified the robustness and scalability of

Agent Factory, demonstrating it constitutes a viable platform for PDA devices.

7. RELATED WORK

The creation of Agent Prototyping Environments (APEs) has proved a rich vein of research over the last decade. In fact, the AgentLink website currently lists 107 agent software applications that are available for download[1]. In this paper, we compare Agent Factory (AF) with some of the more pre-eminent offerings.

By far the closest offering to AF is LEAP [3]. This environment is a synthesis of two earlier Agent Prototyping Environments: JADE [2], and ZEUS [17]. LEAP delivers a FIPA-compliant Agent Platform through the JADE component, and tool-based support for agent development through the ZEUS component. Specifically, LEAP includes a re-factored version of the JADE code base that is compliant with J2ME [21] and extends this code base to include the additional ZEUS functionality. This enables developers to create agents using the ZEUS development environment and then to deploy agents using the revised JADE run-time environment.

A variety of key differences exist between the approach taken with LEAP and that taken with AF. The first arises from the structure of the agent program. ZEUS uses Frames for representing facts and includes a custom language for specifying tasks. This contrasts with AF, which is founded upon multi-modal temporal logic. In addition, ZEUS is oriented towards the fabrication of individual agent programs. As a result, reuse is supported through the definition of Tasks. However, unlike AF, there is no concept of an agent design and there is no support for reuse through inheritance. Conversely, in terms of interoperability with other platforms, LEAP is 100% FIPA-compliant. This contrasts with Agent Factory where efforts at ensuring FIPA-compliance are still ongoing.

A second competing Agent Prototyping Environment is JACK Intelligent Agents [4]. JACK offers a cohesive development environment that supports the fabrication of JACK agents programs. These programs are defined using the JACK Agent Language, an extension to Java that provides constructs that represent agent-oriented features. From the run-time environment perspective, JACK suffers from various drawbacks, namely: it does not support agent mobility, and it is not FIPA-compliant (although FIPA-compliance can be built on top of JACK).

Finally, we consider FIPA-OS [12], an Open Source project, which built the first FIPA-compliant agent platform. FIPA-OS is fundamentally a library of Java classes that implement the FIPA standards. Developers work with FIPA-OS by extending various base classes implementing any application-specific logic. With respect to Agent Factory, FIPA-OS does not directly support the implementation of BDI-type agents, nor does it include tool-based support for the construction or visualisation of agents. Design reuse is supported through the sub-classing of Java classes.

[1] Value was taken from http://www.agentlink.org/ as of the 6/11/2002.

Table 1: A Comparison of Agent Prototyping Environments

	AF	LEAP	JACK	ZEUS	JADE	FIPA-OS
BDI	v		v			
Mobility	v	v			v	v
White Pages	v	v		v	v	v
Yellow Pages	v	v		v	v	v
FIPA Compliance		v		v	v	v
Fabrication Mode	Design	Instance	Design	Instance	Design	Design
Inheritance	v		v		v	v
Construction	Graphical	Graphical	Graphical	Graphical	None	None
Visualisation	Graphical	Graphical	None	Graphical	None	None
Integrated Methodology	v	v		v		

To conclude this review, we present a brief comparison of Agent Factory with respect to the APEs discussed above. This has been realised through the selection of various evaluation criteria. From a deployment perspective, we were concerned with:

- **BDI**. Whether or not the environment delivers direct support for the fabrication of BDI-agents.
- **Mobility**. Whether or not the APE provides support for agent migration.
- **White Pages**. Whether or not the APE includes an agent naming service (i.e. a service by which an agent is able to lookup the physical addresses of other agents).
- **Yellow Pages**. Whether or not the APE includes support for the advertising of agent's services.
- **FIPA Compliance**. Whether or not the APE is FIPA-compliant.

Further, from a development perspective, our evaluation was concerned with:

- **Fabrication Mode**. Whether the environment promotes the fabrication of agent designs (programs that can be used to realise multiple agents from a single design), or agent instances (programs that realise only one agent per design).
- **Inheritance**. Whether the APE supports the rapid prototyping of agents through inheritance.
- **Construction**. Whether the APE provides tool-based support for the fabrication of agents, and what form it takes (graphical, or none).

- **Visualisation**. Whether the APE provides tool-based support for the visualisation of agent behaviour enactment, and what form it takes (graphical, or none).
- **Methodology**. Whether methodological support has been defined to promote structure use of the APE.

The results of this comparison are presented in table 1.

8. CONCLUSION

In this paper, we have introduced Agent Factory, a cohesive framework for the development and deployment of agent-oriented applications. This framework has been designed to support the deployment of agent-oriented applications on Java-compliant devices. In particular, it includes a visually intuitive development environment that promotes design reuse, has strong links with Agent UML, and which delivers a rich set of tools to support the development and debugging of agent designs.

The system provides a rich coupling between an Agent Development Methodology, the *Agent Fabrication Process (AFP)*, and the cohesive Agent Factory Toolset. The AFP oversees the controlled and appropriate temporal invocation of the component tools providing a structure and frame for the agent design and development process. We show how the Run-Time Environment delivers support for mobile intelligent agents. We illustrate the usage, robustness, and scalability of the system via the WAY Case Study and finally situate our work within the broader research landscape.

9. REFERENCES

[1] Bauer, B., Muller, J., Odell, J., Agent UML: A Formalism for Specifying Multiagent Interaction. Agent-Oriented Software Engineering (Paolo Ciancarini and Michael Wooldridge eds), Springer, Berlin, pp 91-103, 2001.

[2] Bellifemine, F., Poggi, A., Rimassa, G., JADE – A FIPA-compliant agent framework, in Proceedings of the 4th International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agents (PAAM), London, UK, 1999.

[3] Berger, M., Bauer, B., Watzke, M., A Scalable Agent Infrastructure, 2nd Workshop on Infrastructure for Agents, MAS and Scalable MAS. Autonomous Agents'01, Montreal, 2001.

[4] Busetta, P., Ronnquist, R., Hodgson, A., and Lucas, A., JACK Intelligent Agents –Components for Intelligent Agents in Java, in AgentLink Newsletter 1, January, 1999.

[5] Collier, R.W., Agent Factory: A Framework for the Engineering of Agent-Oriented Applications, Ph. D. Thesis, University College Dublin, Ireland, 2001.

[6] Colom, J.M., Koutny, M. (eds), Applications and Theory of Petri Nets, Proceedings of the 22nd International Conference ICATPN 2001 Newcastle upon Tyne, UK, June 25-29, LNCS 2075, Springer-Verlag Publishers, 2001.

[7] CVS Home Page, http://www.cvshome.org/

[8] Cost, S., Chen, Y., Finin, T., Labrou, Y., Peng, Y., Using Colored Petri Nets for Conversation Modelling, Issues in Agent Communication – LNAI 1916 eds G. Goos, J. Hartmanis, J. van Leeuwen, Springer-Verlag, 1999.

[9] DeLoach, S.A., Wood, M.F., Sparkman, C.H., Multiagent Systems Engineering, International Journal of Software Engineering and Knowledge Engineering, Volume 11, No. 3, pp 231-258, 2001.

[10] Dennett, D.C., The Intentional Stance, The MIT Press, MA, ISBN 0 262 54053 3, 1987

[11] The FIPA Website, http://www.fipa.org/.

[12] The FIPA-OS Website, http://fipaos.sourceforge.net/.

[13] Juan, T., Pearce, A., Sterling, L., Extending the GAIA methodology for complex open systems, Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS-2002), Bologna, July 15-19, 2002

[14] The K Teams Website, http://www.k-team.com/.

[15] Lowen, T.D., O` Hare, P.T., O` Hare G.M.P, Mobile Agents point the WAY: Context Sensitive Service Delivery through Mobile Lightweight Agents. Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS-2002), Bologna, July 15-19, 2002

[16] Martin, F. J., Plaza, E., Rodríguez-Aguilar, J. A., Conversation Protocols: Modeling and Implementing Conversations in Agent-Based Systems, Issues in Agent Communication - LNAI 1916 (eds G.Goos, J.Hartmanis, J. van Leeuwen), Springer-Verlag, 1999.

[17] Nwana, H., Ndumu, D., Lee, L., Collis, J., ZEUS: A Tool-Kit for Building Distributed Multi-Agent Systems, in Applied Artifical Intelligence Journal, Vol. 13 (1), p129-186, 1999.

[18] O'Hare, G.M.P., Agent Factory: An Environment for the Fabrication of Distributed Artificial Systems, in O'Hare, G.M.P. and Jennings, N.R.(Eds.), Foundations of Distributed Artificial Intelligence, Sixth Generation Computer Series, Wiley Interscience Pubs, New York, 1996.

[19] O'Hare, P., O'Hare G.M.P., and Lowen, T., Far and a way: Context sensitive service delivery through mobile lightweight PDA hosted agents. In Proc. FLAIRS 02, 2002.

[20] Padgham, L., Winikoff, M., Prometheus: A Methodology for Developing Intelligent Agents, Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS-2002), Bologna, July 15-19, 2002

[21] The Sun Java Website, http://java.sun.com/.

[22] Venners, B., Inheritance versus Composition: Which one should you choose? Javaworld,

http://www.javaworld.com/javaworld/jw-11-1998/jw-11-techniques.html

[23] Wooldridge, M., Jennings, N.R., Kinny, D., The Gaia Methodology for Agent-Oriented Analysis and Design, in Journal of Autonomous Agents and Multi-Agent Systems, Kluwer Academic Publishers, Volume 3, pp 285-312, 2000