Agent Factory Micro Edition: A Framework for Ambient Applications

C. Muldoon[1], G.M.P. O'Hare[2], R. Collier[1], and M.J. O'Grady[2]

[1] Practice & Research in Intelligent Systems & Media (PRISM) Laboratory, School of Computer Science and Informatics, University College Dublin (UCD), Belfield, Dublin 4, Ireland
{conor.Muldoon, rem.collier}@ucd.ie
[2] Adaptive Information Cluster (AIC), School of Computer Science and Informatics, University College Dublin (UCD), Belfield, Dublin 4, Ireland
{gregory.ohare, michael.j.ogrady}@ucd.ie

Abstract. Ambient Intelligence represents a vision of the future whereby the world will be saturated with embedded electronic devices that are sensitive and responsive to people. This technology will combine the concepts of intelligent systems with that of pervasive computing. Intelligent agents of varying capabilities will provide the foundations for many applications within this domain. As a means of achieving this objective a framework - Agent Factory Micro Edition (AFME) has been developed to enable the creation of agent-based applications on computationally constrained devices such as cellular digital mobile phones. It has been specifically designed to tackle the performance and memory footprint issues associated with executing intentional agents on mobile devices.

1 Introduction

Ambient Intelligence (AmI) represents the convergence of pervasive and intelligent systems, envisaging a world embedded with sensors and other electronic devices that communicate in a seamless and intuitive manner [1]. How to actually embed intelligence into computationally restricted artifacts and environments remains a key challenge. In an effort to address this issue a framework - Agent Factory Micro Edition (AFME), which supports the deployment of intelligent agents in AmI environments, has been developed. AFME is broadly based upon a preexisting framework that supports a structured approach to the development and deployment of agent-oriented applications, namely Agent Factory [2] [3].

Traditionally, applications incorporating Agent Factory have been deployed in workstation environments. The framework was implemented using Java 2 Standard Edition (J2SE). AFME differs from the original version in several ways. Many of these differences are because the system is based on the Constrained Limited Device Configuration (CLDC) Java platform augmented with the Mobile Information Device Profile (MIDP) rather than J2SE. CLDC and MIDP constitute a subset of the Java 2 Micro Edition (J2ME) Java specification.

A significant difference between (1) AFME and the original version of the framework, and (2) AFME and other embedded platforms, concerns the style in

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part III, LNCS 3993, pp. 727 – 734, 2006.

which the object-oriented components have been written. Specifically, AFME has been developed without the use of accessor (get/set) methods. Accessor methods are really just an elaborate way of exposing an object's internal state and writing code in an alternative style considerably improves the maintainability of the software. It prevents alterations to the types of an object's internal attributes from propagating throughout the code. By minimizing code duplication the footprint of the system is reduced: a critical issue in embedded systems. In [4] the authors prove that any object-oriented system can be rewritten without the use of accessors[1] or exposing an object's state. Thus the developer is assured that there is always an alternative to the accessor approach.

AFME provides support for agnostic communication. The components of applications developed using the framework interact without directly referencing each other and are prevented from containing inter-component dependencies. This coerces developers into adopting a structure to their code that promotes reuse and modularity.

2 Related Research

Considerable research has been invested into the development of Multi-Agent Systems that operate on mobile devices. 3APL-M [5] is a platform that enables the fabrication of agents using the Artificial Autonomous Agents Programming Language (3APL) [6] for internal knowledge representation. Its binary version is distributed in J2ME and J2SE compilations. 3APL provides programming constructs for implementing agents' beliefs, goals, basic capabilities, and a set of practical reasoning rules.

The Foundation for Intelligent Physical Agents (FIPA) is an autonomous standards committee with the objective of facilitating interoperability among agent frameworks. It has ratified an international Agent Communication Language (ACL) to support inter-agent communication. MicroFIPA-OS is a minimised version of the FIPA-OS agent toolkit developed for mobile devices [7]. It provides support for the ACL standard along with yellow and white page services[2]. The system can run in minimal mode whereby agents don't use task and conversation managers. The platform is entirely embedded; however it is recommended that only one agent should operate on low specification devices.

The Light Extensible Agent Platform (LEAP) [8] is a FIPA compliant agent platform capable of operating on both fixed and mobile devices with various operating systems. LEAP extends the Java Agent DEvelopment Framework (JADE) with a set of profiles that allow it to be configured for various Java Virtual Machines (JVMs). The platform is modular and contains components for managing the life cycle of the agents and controlling the motley of communication protocols. The platform is split into several agent containers - one for every device or workstation used.

Though sharing the same broad objectives of these projects, AFME differs in a number of ways. The system has been developed without the use of accessor methods

[1] The Law of Demeter is not limited to accessor methods. It is concerned with all methods that are not closely related to the object or class in question.

[2] Yellow pages enable agents to advertise the services they provide. White pages are used to find the address of an agent if the unique agent name is known.

or exposing an object's state. Writing code in an alternative style simplifies the message passing structure between objects, minimizes duplicated code, and in general reduces the footprint of the software. This is why AFME with a jar size of 86k is probably the smallest FIPA compliant deliberative agent platform in the world.

AFME differs from LEAP and MicroFIPA-OS in that AFME agents have the capability to reason about goals and intentions. Goals are essential for practical reasoning because they enable an agent to identify the purpose of a particular task. By abstracting this information we provide agents with a mechanism to recover from failures and to opportunistically take advantage of unexpected events or possibilities as they become available. Agents are resource bounded and will be unable to achieve all of their goals even if the goals are consistent. The subset of goals that an agent commits resources to achieving constitutes the agent's intentions. 3APL-M does provide rules for practical reasoning however AFME offers significant maintainability advantages over 3APL-M due to the object-oriented style in which it has been written and the support that it provides for agnostic communication (Section 5).

3 The Framework

AFME is based on Agent Factory, a preexisting FIPA compliant framework for the fabrication of a type of software agent that is: autonomous, situated, socially able, intentional, rational, and mobile [3]. It expresses an agent's internal state through the mentalistic notions of belief and commitment. Rules that define the conditions under which commitments are adopted are used to encode an agent's behaviour. This approach is consistent with the Belief-Desire-Intention (BDI) model of agency [10]. The framework is comprised of four-layers that deliver: a programming language, a run-time environment, an integrated development environment, and a development methodology. A detailed description of these components may be found in [3].

The differences between the specifications of the J2SE and J2ME have had a major impact on the design of the object-oriented components of AFME. A complete reengineering of the original system was necessitated, as it contained dependencies on APIs that do not exist within either CLDC or MIDP.

Although there are significant differences in the infrastructure used to build the platforms, AFME and the standard version of the system are consistent in terms of their support for executing agents written in the Agent Factory Agent Programming Language (AFAPL). Communication on both systems is FIPA compliant and thus interoperable. Agents on an AFME platform can migrate to a standard platform and vice versa. This consistency enables the developers of AFME applications to use the preexisting integrated development environment, methodology, and compiler for the creation of agent designs for constrained devices. These agent designs are used by the AFME compiler to generate the requisite MIDlet for the application.

4 Managing Complexity in AFME

Software systems have a greater degree of complexity for their size because if two parts of a program are the same they are placed within a subroutine [10]. Thus each

part of a computer program will be unique. In this respect, software differs profoundly from all other types of human construction in that it does not contain any repeated elements. Avoiding the use of accessor methods makes this ineluctable complexity of software easier to manage by reducing the entropy or restricting the movement of information within the system. It prevents the code dropping back to a semi-procedural system in which the developer has global access to information and an object's attributes move around a number of classes. Furthermore, maintainability is increased as an object's encapsulation is not violated, the types of its attributes are not exposed, and its internal structure remains hidden [11].

When writing code the programmer should view objects as a group of cooperating entities capable of performing tasks and passing messages to one another rather than as a data structure that contains functions or methods to alter its internal state. Objects should be conceptualized in terms of their capabilities and thus the focus of design should be on the messages that are passed between objects, not on how their internal structure has been implemented. The user of an object should never ask for data to perform some action; rather it should ask the object to do the work on their behalf [11]. The consequences of this are that messages flow within the system not data.

To see the negative impact that accessor methods have on an object-oriented system consider a class with a *getAttribute()* method that returns an *int*. Imagine this *getAttribute()* method was called 500 times by external users of the class. If at a later stage a seemingly trivial alteration were made such as changing the attribute's type from an *int* to a *long* a significant amount of code would have to be rewritten. This is because the developer would be coerced into making additional modifications at numerous locations where the method was invoked. Conversely, if the object had been designed to enable the external users to ask the object to perform the relevant work on the attribute on their behalf, this alteration and any related bugs would have been localized. The developer would only have to make a small number of modifications rather than hundreds of them.

Writing code in this manner has implications for how plug-ins may be developed. Classes that use accessor methods when extending the functionality of other classes are not really plugins in the true sense of the word. Consider a television whose functionality has been extended by a video recorder. If at a later stage the television's internal attributes are altered, for example a new flat screen television was bought, the video recorder does not have to be rebuilt. This is because it does not contain any dependencies on the internal components of the television. Now imagine that someone opened up the original television and used some of its parts in building a different device such as a microwave oven. When the television is replaced the microwave has to be rebuilt if it is to reflect the new changes of the television's internal attributes. What matters with plug-in technology is that the communication structures between the object and its extensions remain consistent. Any alteration to the internal attributes of the object or the extension is localized because these entities are only concerned about how they communicate with each other. When developing code that needs to be extensible, the manner in which it communicates with the plug-ins must be explicitly designed into the object. The video recorder could not have extended the television if the television were not specifically designed to have a socket for the video cable or the cables of other peripheral devices that communicate consistently. Otherwise the video recorder would have to access the internal components of the television.

When developing multi-agent systems for embedded devices we attempt to condense the same amount of functionality of the platforms developed for desktop machines into a much smaller space. This increases the complexity of the software for its size. Avoiding the use of accessors enables a developer to manage this increased complexity in that the code is structured in a manner that prevents maintainability problems. Accessor methods effectively make an object's internal attributes global therefore any alterations to the attributes will also be global. This causes precisely the type of maintainability problem that object-oriented programming is supposed to prevent. One of the tenets of good object-oriented development is that it is possible, by protecting an object's internal state, to radically alter the internal structure of a class without making changes to the users of the class. The developers of other embedded agent platforms that extensively use accessors have not adopted this concept. In contrast, AFME does not contain accessor methods[3]. It has been our experience that developing in alternative style usually improves performance because in most cases the number of method invocations within the system is reduced.

5 Supporting Agnostic Communication

AFME delivers support for the creation of BDI agents that follow a sense-deliberate-act cycle. To facilitate this process a number of system components have been created. Developers extend these components when building their applications. These components are perceptors, actuators, modules, and services. Perceptors and actuators enable agents to sense and to act upon their environment respectively. Modules represent a shared information space between actuators and perceptors. They are used when actuators and perceptors must communicate with an object instance internal to the agent, for example an actuator may pass a message to affect the state of an object instance and a perceptor may perceive that resultant effect. Services also represent a shared information space but between agents rather than actuators and perceptors. Services are used when information is passed between agents for example the local Message Transport Service.

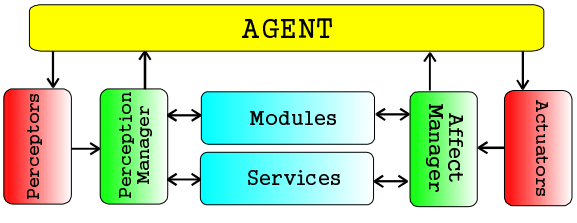

Fig. 1. Communication Structure of system components in AFME

To improve reuse and modularity within the system these entities are prevented from containing direct object references to each other and the agent class. Rather than passing messages directly they interact via perception and affect managers (Fig. 1).

[3] No accessor methods have been written in the application code. Accessors are invoked on objects that form part of the core Java APIs where no alternatives are provided.

This ensures that communication between the components is agnostic. The messages that are passed between the components are in the form of first order structures. First order structures provide a symbolic representation of the information content and ensure that messages passed do not expose internal details of the message senders. Actuators and perceptors developed to interact with a service in one application can be used without making any coding alterations to interact with a module in a different application and vice versa. The implementation of modules or services can be completely altered without having to modify or recompile the actuators and perceptors. A completely different class could even be used to provide the functionality. Additionally, the same service or module may be used within two different applications to interact with a different set of actuators and perceptors.

The system components of AFME are interchangeable because they interact without directly referencing one another. They contain dependencies on the first order structure class and the affect and perception managers, which are generic components of the system. They do not contain dependencies on each other. When a module or service is created they are associated with a uniquely identifiable name. Actuators and perceptors use this name to indicate the target object for a particular message. They call the appropriate method on the affect and perception managers. The name is resolved to a module or service instance and the message is forwarded on appropriately.

6 Message Transport Service

The Message Transport Service of AFME (Fig. 2) had to be changed considerably from the original design. This was because our local GPRS and 3G service providers have a firewall operating thus preventing incoming socket connections and 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

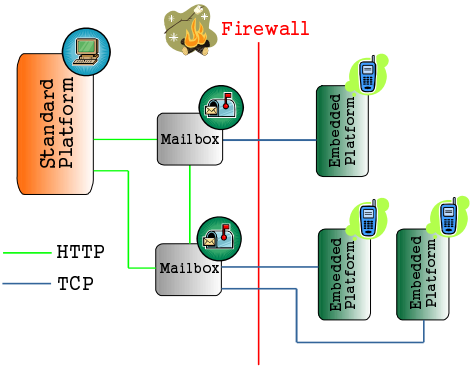

Fig. 2. AFME Message Transport Service

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.

7 Migration

Within AFME support is provided for weak migration. Any classes required by the agent must already be present at the destination. This is because CLDC does not contain an API for introspection[4] and is prevented from dynamically loading foreign objects. To facilitate the migration process a similar approach has been adopted to that of the Message Transport Service. Agents migrate to a migration server where they wait for a connection from their destination. 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 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. Agents maintain beliefs about where they can download the requisite commitment rules for a particular environment. When an agent migrates it sends this information to the migration server before it is transferred. When the destination platform connects to the migration server its type is specified. The migration server uses the destination type and the information sent by the agent to obtain the requisite commitment rules. The rules are then added to the agent design. This enables a transparent migration process. The agent need not be aware of the type of environment it is migrating to before it migrates.

8 Conclusion

Intelligent agents encapsulate certain characteristics that make them suitable for creating ambient applications. Their autonomous nature, ability to react to external events as well their capability to be proactive in fulfilling their objectives make them apposite for operating in complex and dynamic environments. Embedded devices are, almost by definition, computationally constrained. Thus the goal of delivering intelligent agents on such devices is one fraught with difficulty, since agent platforms often have a large footprint. AFME is a framework that been developed to address this issue. The design decisions taken for the construction of the platform have been described in this paper. The approach taken significantly improves the maintainability of the software when compared to other embedded agent platforms currently available.

Complex software solutions will be required if the vision of AmI is to be fulfilled. Embedded intelligent agents offer one promising approach to realising the *intelligence* essential to AmI. Through the development of AFME we have demonstrated that such

[4] Local classes packaged within the application jar file can be dynamically loaded.

an approach is feasible. AFME is an open source project and is freely available for download from the Agent Factory SourceForge web site under the terms of the GNU Lesser General Public License [12].

References

1. Aarts, E, Marzano, S. (editors), The New Everyday: Views on Ambient Intelligence, 010 Publishers, Rotterdam, The Netherlands, 2003.
2. O'Hare G.M.P., Agent Factory: An Environment for the Fabrication of Multi-Agent Systems, in Foundations of Distributed Artificial Intelligence (G.M.P. O'Hare and N. Jennings eds) pp. 449-484, John Wiley and Sons, Inc., 1996.
3. Collier, R.W., O'Hare G.M.P., Lowen, T., Rooney, C.F.B., (2003), Beyond Prototyping in the Factory of Agents, in Multi-Agent Systems and Applications III: Proceedings of the 3rd Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'03), Prague, Czech Republic, Lecture Notes in Computer Science (LNCS 2691), Springer-Verlag.
4. Lieberherr, K.J., Holland, I, Riel, A.J., Object-oriented programming: An objective sense of style, in Object Oriented Programming Systems, Languages and Applications Conference, in special issue of SIGPLAN notices, number 11, pages 323-334, San Diego, CA, 1988.
5. 3APL-M: Platform for Lightweight Deliberative Agents: http://www.cs.uu.nl/3apl-m/
6. Birna van Riemsdijk, M.D., Dignum, F., Meyer, J.J., A Programming Language for Cognitive Agents: Goal Directed 3APL. Proceedings of the First Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS), Melbourne, 2003.
7. Tarkoma, S., Laukkanen, M., Supporting Software Agents on Small Devices: Proceedings of the first international joint conference on Autonomous Agents and Multi-Agent Systems (AAMAS), Bologna, 2002.
8. Berger, M., Rusitschka, S., Toropov, D., Watzke, M., Schichte, M., Porting Distributed Agent-Middleware to Small Mobile Devices: Proceedings of the Workshop on Ubiquitous Agents on embedded, wearable, and mobile devices held in conjunction with the joint conference on Autonomous Agents and Multi-Agent Systems (AAMAS), Bologna, 2002.
9. Rao, A.S., Georgeff, M.P.: Modelling Rational Agents within a BDI Architecture. In: Principles of Knowledge Representation. & Reasoning, San Mateo, CA. 1991.
10. Brooks, F.P., No Silver bullet: Essence and Accidents of Software Engineering, in IEEE Computer, Vol. 2 No. 4, April 1987, pp. 10-19.
11. Holub, A., Building user interfaces for object-oriented systems, Part 1.
12. http://www.javaworld.com/javaworld/jw-07-1999/jw-07-toolbox_p.html
13. Agent Factory SourgeForge repository. http://agentfactory.sourceforge.net/