A Model Driven Component Agent Framework for Domain Experts

A thesis submitted for the degree of
Doctor of Philosophy

Gaya Buddhinath Jayatilleke B.Sc (Hons.), M.SoftSysEng,
School of Computer Science and Information Technology,
Science, Engineering, and Technology Portfolio,
RMIT University,
Melbourne, Victoria, Australia.

17th March, 2007

Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; and, any editorial work, paid or unpaid, carried out by a third party is acknowledged.

Gaya Buddhinath Jayatilleke
School of Computer Science and Information Technology
RMIT University
17th March, 2007

Acknowledgments

The four year journey, which this thesis is a result of, would not have been possible without the many who supported me. It is difficult to express my gratitude to my two supervisors, Prof. Lin Padgham and A/Prof. Michael Winikoff, whose help and guidance kept me focused. The knowledge I gained from them over the last four years is priceless. I thank them both.

I thank my parents, who sacrificed a lot in their lives for my education. I could not have come this far if not for their loving care. I thank my wife, whose patience, understanding and sacrifices gave me the freedom to concentrate on the research.

My research was supported by various grants and organizations including the Bureau Of Meteorology (BoM), Agent Oriented Software (AOS) and Australian Research Council (ARC). I thank BoM for providing me with a domain to test my research and especially all who participated in the user study. I thank AOS for allowing me to use the JACK agent language in my research work. I thank ARC for the grants (Linkage Grants LP0347025, LP0453486) that supported my research work.

I thank RMIT University and the Department of Education, Science and Training (DEST) for providing me with a scholarship, that allowed me to pursue my dream of reading for a PhD. I think my colleagues in the RMIT Agents Group for their support in providing feedback on my work and encouragement. I can not forget everyone at the School of Computer Science and Engineering, who were part of the great environment I was lucky to be in. I thank you all.

Credits

Portions of the material in this thesis have previously appeared in the following publications:

- Gaya Buddhinath Jayatilleke, Lin Padgham and Michael Winikoff, A model driven development toolkit for domain experts to modify agent based systems, in *7th International Workshop on Agent Oriented Software Engineering (AOSE'06)*, Held at AAMAS 2006, May, 2006, Hakodate, Japan (work included in Chapter 5).

- Gaya Jayatilleke, John Thangarajah, Lin Padgham and Michael Winikoff, Component Agent Framework for domain-Experts (CAFnE) Toolkit, demonstration given at the *Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-06)*, May, 2006, Hakodate, Japan (work included in Chapter 4).

- Gaya Jayatilleke, Lin Padgham, and Michael Winikoff. Component Agent Framework for non-Experts (CAFnE) Toolkit. Chapter in *Software Agent-Based Applications and Prototypes*, edited by Rainer Unland, Monique Calisti and Matthias Klusch. Whitestein Series in Software Agents Technologies, Pages 169-196, ISBN 978-3-7643-7347-4, Birkhaeuser Publishing Company, Sept, 2005 (work included in Chapter 4).

- Gaya Buddhinath Jayatilleke, Lin Padgham, and Michael Winikoff. A Model Driven Component-Based Development Framework for Agents. in *International Journal of Computer Systems Science & Engineering*, Vol 20 No 4 July 2005 (special issue on Best of MATES 2004) (work included in Chapter 3).

- Gaya Buddhinath Jayatilleke, Lin Padgham and Michael Winikoff, Towards a Component-Based Development Framework for Agents, *Second German Conference on Multiagent System Technologies* (MATES), 2004, Erfurt, Germany (work included in Chapter 3).

- Gaya Buddhinath Jayatilleke, Lin Padgham and Michael Winikoff, Towards a Component-Based Agent Development Workbench for non-Experts, (short paper), *6th NODe Young Researchers Workshop 2004*, Sep, 2004, Erfurt, Germany.

This work was supported by the Australian Research Council (Linkage Grants LP0347025, LP0453486).

The thesis was written in the `TeXnicCenter` editor on Microsoft Windows XP Professional, and typeset using the LaTeX 2_ε document preparation system.

Dedication

To my mother, Jayanthi Jayatilleke, who has been the wind beneath my wings!

Contents

List of Figures

Abstract

Industrial software systems are becoming more complex with a large number of interacting parts distributed over networks. Due to the inherent complexity in the problem domains, most such systems are modified over time to incorporate emerging requirements, making incremental development a suitable approach for building complex systems. In domain specific systems it is the domain experts as end users who identify improvements that better suit their needs. Examples include meteorologists who use weather modeling software, engineers who use control systems and business analysts in business process modeling. Most domain experts are not fluent in systems programming and changes are realised through software engineers. This process hinders the evolution of the system, making it time consuming and costly. We hypothesise that if domain experts are empowered to make some of the system changes, it would greatly ease the evolutionary process, thereby make the systems more effective.

Agent Oriented Software Engineering (AOSE) is seen as a natural fit for modeling and implementing distributed complex systems. With concepts such as goals and plans, agent systems support easy extension of functionality that facilitates incremental development. Further agents provide an intuitive metaphor that works at a higher level of abstraction compared to the object oriented model. However agent programming is not at a level accessible to domain experts to capitalise on its intuitiveness and appropriateness in building complex systems.

We propose a model driven development approach for domain experts that uses visual modeling and automated code generation to simplify the development and evolution of agent systems. Our approach is called the **C**omponent **A**gent **F**ramework for domai**n**-**E**xperts (CAFnE), which builds upon the concepts from the SMART definition framework [d'Inverno and Luck, 2001], Prometheus design methodology [Padgham and Winikoff, 2004] and Model Driven Architecture (MDA) [Kleppe et al., 2003]. CAFnE enables domain experts to work with a graphical representation of the system, which is easier to understand and work with

than textual code. The model of the system, updated by domain experts, is then transformed to executable code using a transformation function. CAFnE is supported by a proof-of-concept toolkit that implements the visual modeling, model driven development and code generation. We used the CAFnE toolkit in a user study where five domain experts (weather forecasters) with no prior experience in agent programming were asked to make changes to an existing weather alerting system. Participants were able to rapidly become familiar with CAFnE concepts, comprehend the system's design, make design changes and implement them using the CAFnE toolkit, within a relatively short period of time of two to three hours.

Chapter 1

Introduction

Everything should be made as simple as possible, but not simpler.
- Albert Einstein

Computer application domains are increasing in complexity. Software used in weather modeling and in robot control systems are examples of such complex systems. Building and maintaining these systems demands new approaches in software engineering to manage the inherent complexity.

A key characteristic of complex systems is that they are continuously modified as new or changed requirements emerge. In most complex domains, it is the domain experts who identify improvements as end users and initiate the evolutionary process[1]. Domain experts are normally not fluent in systems programming and changes are realised through software engineers. The ability of domain experts to make some of the improvements can greatly enhance the system evolution while reducing the time and cost of the change cycle.

Agent oriented software engineering (AOSE) is considered a natural fit for modeling and implementing distributed complex systems [Jennings, 2001]. Characteristics of agent systems such as autonomy and goal-plan based execution help manage the inherent complexity. The agent model is intuitive and works at a higher level of abstraction compared to the object oriented model. While agent systems support easy extension of functionality with mechanisms such as goal and plan addition, agent programing is not at a level accessible to domain experts.

Our work explores the possibility of empowering domain experts at the conceptual and tool support level to allow them to make modifications to agent systems. We attempt to

[1]For example a meteorologist who uses a weather monitoring system and wants to add a new feature.

provide a clear set of building blocks easily comprehensible by domain experts and with the right amount of expressiveness in specifying an agent system. We investigate the nature of tool support required to assist the domain experts in understanding a complex system and easily making modifications. Our intention is not to have domain experts building and maintaining an agent system at all possible levels. The aim is to empower domain experts to make common changes such as adding new agents and editing plans, which are part of the evolutionary process of an agent system.

1.1 Case for Domain Expert Programming

Industrial software systems are becoming more complex with a large number of parts and interactions between them [Jennings, 2001]. In many cases the parts are decoupled and distributed over a network. Software used in telecommunications, business process modeling and manufacturing execution systems are examples of such complex systems. Software engineering has been devising approaches and techniques to tackle the growing complexity in systems such as improvements in development methodologies, component based software engineering [Heineman and Council, 2001] and aspect oriented programming [Kiczales et al., 1997].

A common dilemma faced by software engineers in building complex systems is the lack of clear requirements and domain knowledge needed to come up with a detailed design of the system. Factors such as the inherent complexity of the problem domain, number of stakeholders involved and the long duration of development often leads to systems that deviate from the desired functionality. New requirements leading to such changes are often discovered by end users after the system is deployed in the actual environment. Additionally, errors not captured in the testing phase are also identified by end users, once the system is in use[2]. Hence a more practical approach is an iterative development strategy that allows the evolution of a complex system by modifying it to incorporate new requirements.

In domain specific systems it is the domain experts who find new requirements and modifications, as end users. For example a meteorologist working with a weather alerting

[2]According to [van Vliet, 2000] *perfective* and *adaptive* changes make up seventy five percent of system maintenance tasks with total maintenance cost being at least fifty percent of the total life cycle cost. Perfective changes are made to improve a system such as adding new features and adaptive changes are used to make a system function in new environments. Two other change types identified by [van Vliet, 2000] are *corrective* changes made to repair defects and *preventive* changes made proactively to improve the maintainability of the system.

system may find the need for a new type of alert. We view a domain expert as an individual who is proficient in a given domain with or without formal training in computer programming. Examples of domain experts include scientists, meteorologists, architects, engineers etc. They are different to the typical 'novice programmer' [Smith et al., 2000] (see section 1.3) in that domain experts are able to understand a system at the logical level due to their maturity in domain knowledge and experience in using domain specific systems. However becoming proficient in software engineering to change a system at the implementation level is not a plausible option for many domain experts. Hence the usual approach in making changes is to direct the new requirements to software engineers for making the necessary updates. Such a process hinders the evolution of the system and is both time consuming and costly.

We hypothesise that if domain experts are empowered with a suitable development environment where they can make modifications to an existing system it could greatly ease the evolution of the system. While domain experts may not be able to make changes at all levels, we find that with the right level of support they are more likely to be able to change domain specific functions of a system as opposed to platform or system architecture related changes. For example, an engineer is likely to devise a new manoeuvre for a robot using its basic actions but not be able to write a driver to control a new tool attached to the robot, without the knowledge of low level programming. In order to achieve this, domain experts need support in comprehending a system design and an intuitive programming approach at the implementation level. We also require a conceptual model for defining an agent system and tool support for the modification process. Software engineering practices, including programming environments, currently used in building complex systems are not at a level accessible to domain experts. While there are domain specific programming environments (such as Matlab and AutoCAD used in engineering), they are only applicable within a narrow scope compared to general purpose programming languages.

1.2 Agent Oriented Software Engineering

An agent is viewed as an autonomous piece of software characterized by properties such as being reactive, proactive and showing social behaviour [Wooldridge, 2001]. A group of such agents is used to implement a given system. Agent Oriented Software Engineering (AOSE) proposes an approach for modeling and implementing complex systems [Jennings, 2001]. For example the agent metaphor has been found to be a natural fit in modeling business processes in such industries as insurance and packaging [Munroe et al., 2006].

AOSE has been able to bring in powerful concepts from AI such as goals, plans, learning and planning into building robust systems. Further it is argued that characteristics of agent systems cater better to the development needs of distributed complex systems compared to the object oriented approach [Odell, 2002]. Examples of such characteristics include the asynchronous processing of data, distributed nature, self invocation of methods and coarse grained encapsulation of functions. The encapsulation found in agents is specifically important as it supports the evolutionary process by localising changes to an agent or a group of agents. Changes to agent internals are also facilitated through architectures such as the Belief-Desire-Intention (BDI) [Rao and Georgeff, 1995] architecture with a goal-plan based agent design. For example, consider an agent system that monitors a smart house. Initially the agents are provided with plans to notify the authorities and call the owner when an intruder is detected. If there is a new requirement to light up the area of the house where the intruder is, this can be added as a new plan for handling the event of intruder-detection, without affecting the existing functionality.

The notion of an autonomous agent has been found to be an intuitive metaphor for modeling dynamic systems compared to the passive object metaphor [Repenning, 1993]. Its intuitiveness in modeling business processes has been shown to be useful in attracting customer interest through the use of concepts such as agent roles that map well to business roles [Munroe et al., 2006]. Agent programming platforms have allowed the use of such abstract concepts at the implementation level with a close match between intuitive design artifacts and implementation constructs. For example, goals, plans and beliefs are some of the implementation constructs found in agent programming languages such as JACK [Busetta et al., 1998] and Jadex [Pokahr et al., 2005] that follow the BDI architecture.

The natural fit and the intuitive nature of the model make agents a suitable programming paradigm for domain experts working in complex domains. However existing AOSE tools and agent programming languages are developed for experienced programmers and are not suited for domain experts. For example, most agent languages are text based and are implemented as extensions to existing programming languages such as object oriented or logic programming. Hence a domain expert requires knowledge of both the host language and agent programming principles. Even though there is considerable tool support in AOSE for designing and building systems, such as the JACK Development Environment [AOS, 2005a], they still require a user to understand the syntax and semantics of the underlying textual language.

1.3 Existing Work

Our investigations found several areas of work within software engineering that attempt to conceal the complexities in low level programming and provide an abstract set of constructs that help in system comprehension and development. These include novice programming, component based software engineering (CBSE), model driven development (MDD) and Integrated Development Environments (IDEs) including CASE[3] tools. Each of these areas have their strengths and weaknesses with respect to domain expert programming and we examine them in detail in chapter two.

Novice programming looks at developing programming environments for newcomers to computer programming, including children [Smith et al., 2000]. Most novice programming approaches adopt visual techniques to overcome the difficulties in textual programming. While they highlight important aspects of visual programming, the resulting applications of existing novice programming environments are too simple for domains where agent systems are typically used. End user programming is a special case of novice programming where the users are experienced computer users with specific programming needs, but are not professional programmers [Nardi, 1993]. While our work is similar in nature in that domain experts are end users, devising methods to change complex domain applications is not a primary aim of existing end user programming tools.

Component based software engineering is a promising approach for tackling complexity, widely used in mainstream software engineering [Heineman and Council, 2001]. CBSE provides a mechanism to modularise and encapsulate complex functionality in clearly defined components, which makes development a process of assembling components as opposed to working with textual code. A component based approach can be used to enhance understanding and simplify the development of a system, which are desirable features for domain expert use. However existing component technologies widely used in commercial software development, such as Java Beans and Microsoft COM, work at a finer grained level than we require. Further while these technologies provide an implementation platform for components, they do not provide a domain independent model for defining system components. Hence most adoptions of component based technologies in AOSE attempt to modularise agent platforms whereas our interest is in modularising an agent system that runs on the platform.

Model driven development (MDD) proposes a development strategy of creating various high level models of a system and then performing automated transformations on the mod-

[3]Computer Aided Software Engineering

els to generate the executable code [Frankel and Parodi, 2004]. Model Driven Architecture (MDA) is an initiative of the Object Management Group[4] (OMG) that attempts to standardise MDD for mainstream software development. The MDD approach of working with a relatively abstract specification of functionality, which is then transformed to executable code is a potentially suitable development method for domain experts who are new to systems programming. However existing use of MDD concepts such as MDA are developed with experienced software developers in mind. For example MDA uses UML [Booch et al., 1999] as a modeling language and only generates skeleton code, which requires completion. Model driven development is still new to AOSE and there have been attempts to extend MDA concepts for agents using UML as the underlying modeling language to represent agent concepts [Bauer and Odell, 2005]. However, UML is based on object oriented principles and therefore is not a good match for the agent paradigm.

Integrated Development Environments (IDEs) and CASE tools provide software support to work with various development artifacts such as system designs and source code. Examples of typical IDE functions include editing, visual designing, skeleton code generation and debugging support. While tool support is essential, current IDEs, especially ones used in AOSE, are made for experienced programmers who understand the syntax and semantics of the underlying textual language. For example, the available visual design tools such as the JACK Design Environment while aiding the comprehension of the system design, generates only skeleton code requiring completion by a programmer.

The above approaches for simplifying software development highlight various concepts desirable for facilitating the direct involvement of domain experts in systems evolution. However they have not been utilised within AOSE with a focus on empowering domain experts. Our research builds on MDD concepts to develop a framework for defining, representing and implementing agent systems in a way that allows domain experts to understand and modify the system with only a basic knowledge of agent concepts.

1.4 Research Questions

The research revolves around two main problems: first is finding a representation for an agent system that is understandable by a domain expert while still preserving the characteristics of the agent metaphor. Second is finding a method to facilitate the process of making changes, with a minimum burden on the user with respect to lower level technical details. These two

[4]www.omg.org

problems can be further refined with the following research questions:

1. What are the appropriate building blocks and models that define an agent system? These should be:

   - aligned with the underlying concepts of agenthood
   - sufficiently expressive for developing complex applications
   - intuitive to non-programmers
   - independent of any application domain or implementation platform

2. What is an appropriate representation for visualising an agent system that helps in understanding the system design?

3. How can concepts from Model Driven Development (MDD) be used in generating complete executable from the higher level system models? Using MDD requires the definition of an agent meta-model from which a given agent system can be instantiated.

4. What type of tool support can be given to domain experts to assist in the process of understanding and modifying an existing agent system? The tool needs to incorporate visual modeling and code generation with additional support in areas such as manipulation of models and debugging.

1.5 Research Outcome and Main Contributions

The concepts and methods proposed in this thesis for domain expert programming of agent systems are collectively identified as "**C**omponent **A**gent **F**ramework for domai**n** **E**xperts" or "CAFnE"[5] in its abbreviated form. CAFnE is essentially a model driven development approach for building agent systems that uses visual models and transformations between models to generate executable code. The main contributions of the work are summarised below.

Agent meta model:

[5]Domain experts were originally referred to as non-Experts, meaning non *programming* experts and the resulting framework was initially named Component Agent Framework for non-Experts (abbreviated to CAFnE; pronounced 'caffeine'). We kept the same name, except for changing non-experts to domain experts as it is a more appropriate term for the audience we focus on.

The thesis proposes an agent meta model, that defines the essential building blocks required for modeling and implementing an agent system that follows the BDI principles. The meta model has its roots in the SMART framework [d'Inverno and Luck, 2001], which is a Z based formalisation of software agent concepts.

In order to make the meta model usable by domain experts we first defined a simple agent execution model that is both intuitive and rich in functionality. Based on this simple agent model, we defined a set of agent building blocks that include: goal, event, plan, belief, step, attribute, entity, environment and agent that make up the meta model. These concepts are independent of any application domain and of any programming platform. The internal structures and relationships of these building blocks are defined using XML Document Type Definition[6] (DTD) notation.

Model Driven Development for agent systems

The model driven development defined in CAFnE focuses on providing a process where domain experts are supported in modeling an agent application with the necessary runtime code being derived automatically from the models. This automated generation of executable code frees the domain expert from learning a difficult agent programming language and allows them instead to work with graphical models. It is important to note that our interest is in providing domain experts with a suitable development approach and not inventing a new agent programming language. We used existing agent programming languages as the platform for implementing the executable CAFnE system.

CAFnE's MDD is based on three main models namely Meta model (stated above), System model and Executable model. The System model defines a given system and is represented graphically allowing domain experts to easily understand and make system modifications visually. The System model is a non-executable model not bound to any implementation platform or programming language. CAFnE defines a Transformation function to generate an Executable model from the System model. The Executable model is encoded in an existing programming language, preferably an agent oriented language, such as JACK and Jadex. As the System model is platform independent, using an appropriate transformation function it can be mapped to multiple implementation platforms. CAFnE defines the algorithm for the transformation function and demonstrates how it can be used to transform CAFnE models

[6]XML DTD is a grammar standardised by the World Wide Web Consortium (W3C) for defining the meta-level structure of an XML document. More details could be found on the XML specification at http://www.w3.org/TR/2006/REC-xml-20060816/

into two BDI based agent languages namely JACK and 3APL [Hindriks et al., 1999] with dissimilar programming styles[7].

A key difference between our approach and the one proposed by MDA is that, while MDA hypothesises executable code generation, it has only been successful in generating skeleton code. Our approach generates complete executable code, where domain experts do not need to access the underlying textual code for completing changes made to higher level graphical models. A contributory factor in this regard is the inclusion of an Initialization model in CAFnE. A corresponding model is not found in the MDA framework. The Initialization model defines the initial state of the system with respect to agent and data instances, essential in generating an executable system.

CAFnE toolkit for domain experts

The CAFnE Toolkit is a proof-of-concept toolkit that implements the concepts defined in CAFnE and provides support for domain experts to modify CAFnE based systems. The CAFnE Toolkit is built on top of the Prometheus Design Tool (PDT)[8] and provides a visual environment for working with the various CAFnE models. Domain experts are able to make modifications to a system via the various graphical views available and then use the integrated transformation utility to generate the executable code, and run the system. The current transformation utility includes the transformation function for generating JACK language code. Additionally, the toolkit includes features for debugging, such as checking for conformity of the System model with the Meta model and tracing events at runtime for identifying undesired behaviors. The important aspect here is that users work with the basic building blocks defined by the CAFnE meta model, which aligns with the intuitive agent metaphor. It reduces the learning time required for a domain expert to get familiar with the concepts and then start using them in modifying a system

Evaluation case study

We evaluated the CAFnE Toolkit with a group of five domain experts (meteorologists) who were asked to make several modifications to a sample weather alerting system. The sample system was based on an actual agent system tested at the Bureau of Meteorology, Melbourne and the changes used in the study emulated modifications made to the actual system. All five participants were new to agent programming and only two were experienced object ori-

[7]JACK extends Java with agent concepts whereas 3APL is based on logic programming

[8]www.cs.rmit.edu.au/agents/pdt

ented programmers. They were all able to become familiar with the CAFnE concepts with a 30-40min introduction and then start making the specified changes, despite their skill level variations. The data gathered in the study was analysed using the cognitive model of program understanding by Letovsky et al. [Letovsky, 1986; Littman et al., 1986]. It showed that domain experts with no prior experience in agent programming were able to comprehend an agent system and make moderately complex changes in a short time using the CAFnE Toolkit.

1.6 Structure of the Thesis

In chapter two we look at background material and other research work related to this thesis. This includes agent concepts (especially BDI agents), agent oriented software engineering (AOSE) including various methodologies and toolkits, component based software engineering (CBSE), model driven development (MDD) and programming for novices. The CAFnE meta-model for agents and the model driven development approach are introduced in chapter three. We define each of the CAFnE agent components and show how the Prometheus notation is extended to represent a CAFnE System model. In chapter three we also define the concepts behind the transformation function used to generate executable code from the system model. Chapter four presents the CAFnE toolkit that incorporates the model driven approach described in chapter three. We discuss how we have extended the Prometheus Design Toolkit (PDT) to come up with an implementation tool for domain experts to work with a CAFnE System model. The outcome of the evaluation of the CAFnE approach and the toolkit is discussed in chapter five. Conclusions and future work are presented in chapter six.