A SOFTWARE DEVELOPMENT PROCESS FOR
BDI MULTIAGENT SYSTEMS

AARON HECTOR
B. ENG. (SOFTWARE)

THE UNIVERSITY OF NEWCASTLE,
AUSTRALIA

Thesis submitted for the degree of
DOCTOR OF PHILOSOPHY IN SOFTWARE ENGINEERING

SEPTEMBER 2013
Statement of Originality

The thesis contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. I give consent to the final version of my thesis being made available worldwide when deposited in the University’s Digital Repository, subject to the provisions of the Copyright Act 1968.

Aaron Hector
Acknowledgements

Firstly, I would like to thank my employer and co-workers, for their patience and encouragement as I combined full-time employment with part-time pursuit of a PhD. Without the flexibility granted to me, this simply would not have been possible.

I would like to express my gratitude to my supervisor, Associate Professor Frans Henskens for his patience, understanding and advice throughout the course of this work. I would also like to thank my co-supervisor, Dr Michael Hannaford for his invaluable assistance during this time. My deepest thanks to both of you.

A huge thanks to my wife, Karen, for her encouragement over the past few years. Without your endless, tireless love and support, I would not have been able to do this.

I would also like to thank my fellow PhD students at the University of Newcastle. For everyone in “EF”, thank you for your continued encouragement and enthusiasm. You have all made the experience one to cherish, regardless of the ups and downs faced during the progress of the research. An especially big thanks to Anthony for his ever-present support.
# Table of Contents

Acknowledgements ................................................................. iii

Table of Contents ........................................................................ iv

Abstract ....................................................................................... viii

Chapter 1  Introduction ............................................................... 1

1.1 Overview .............................................................................. 1

1.2 Definitions ............................................................................ 2

1.3 A Process or a Methodology? ............................................... 3

1.4 Motivation ............................................................................ 4

1.5 Practical Research ............................................................... 6

1.6 Research Methodology ....................................................... 7

1.7 Structure of the Thesis ....................................................... 8

Chapter 2  Literature Review ....................................................... 10

2.1 Introduction .......................................................................... 10

2.2 Software Engineering Background ..................................... 10

2.3 Multiagent Systems ............................................................ 14

2.4 Historical Development of Multiagent Systems .................. 24

2.5 Software Processes for Developing BDI Multiagent Systems 33

2.6 Applicability of multiagent systems ................................... 44
4.11 Dependency Tracking ................................................................. 158

4.12 Summary .................................................................................. 164

Chapter 5  NUMAP Support Tool Implementation ............................... 166

5.1 Introduction ............................................................................... 166

5.2 Overview of the NUMAP Support Tool ........................................ 166

5.3 Features of the NUMAP Support Tool .......................................... 167

5.4 Design and Implementation of the NUMAP Support Tool ............... 175

5.5 Difficulties, solutions and alternatives ......................................... 185

5.6 Novel and Notable Aspects ......................................................... 189

Chapter 6  Evaluation ...................................................................... 191

6.1 Introduction ............................................................................... 191

6.2 Selection of an Evaluation Methodology ....................................... 191

6.3 Feature analysis-based evaluation ............................................. 193

6.4 Selection of Features ................................................................ 195

6.5 Overview of Features for Evaluation .......................................... 195

6.6 Feature analysis of NUMAP ...................................................... 202

6.7 Feature analysis of Gaia .......................................................... 207

6.8 Feature analysis of Prometheus ................................................. 212

6.9 Comparison of Evaluation Results ............................................. 216

6.10 Summary .................................................................................. 219
Chapter 7  Conclusion................................................................................................................. 221

7.1  Chapter Synopsis .................................................................................................................. 221

7.2  Summary of Work Presented ................................................................................................. 222

7.3  Contributions of this Work .................................................................................................... 224

7.4  Limitations of the Work ........................................................................................................ 227

7.5  Future Work............................................................................................................................ 228

Appendix A – Classifying Agents ............................................................................................... 230

Appendix B - Mappings Between Phases.................................................................................... 237

Mapping Requirements Phase to Analysis Phase........................................................................ 238

Mapping Analysis Phase to Design Phases .................................................................................. 239

Mapping Organisational Design Phase to Agent Design Phase.................................................. 241

References ...................................................................................................................................... 242
Abstract

Multiagent systems are an increasingly popular approach to software engineering, offering a fundamentally different design technique for software development based around the creation and deployment of autonomous software components.

One of the leading models for the development of agent-based software is the Belief-Desire-Intention (BDI) model. This model provides a method for developing software based around the concepts used in human reasoning. BDI provides not just a useful model for developing software agents, but promises a whole new paradigm for software development, occurring at a higher level of abstraction than traditional techniques.

In order to effectively develop BDI systems, a software development process is required that takes advantage of the concepts and abstractions offered by the BDI model. This thesis presents the Newcastle University Multi-Agent Process (NUMAP), a modular development process for BDI multi-agent systems. The process is practical, usable, and intuitive, operating at a level of abstraction suited to BDI agent development, focusing on the use of concepts associated with human reasoning throughout the development lifecycle.

A support tool for use with NUMAP is also presented, assisting developers as they progress through each phase of the process. This support tool offers code template generation for popular agent implementation platforms, and provides a novel dependency tracking mechanism.

A new evaluation framework is presented for assessing the completeness of development processes for multiagent systems, and NUMAP is assessed in comparison with other leading design processes, showing advantages in terms of its coverage of agent modelling concepts, and in terms of its usage and practicality.
Chapter 1 Introduction

1.1 Overview

The aim of this work is to create a practical, complete software process for developing Belief-Desire-Intention (BDI) [1] multi-agent systems [2]. These systems provide a capable technique for developing complex, decentralised software, comprised of autonomous software components. BDI agents are designed using concepts originating in the philosophical theories of “human action” [3]. The meaning of “BDI” and “multi-agent system”, and the origins of such concepts, will be further defined in Chapter 2.

In order to achieve the aim of this work, namely creating a software process that is useful to real-world developers, a number of tasks needed to be performed.

Firstly, an investigation was carried out into the current state of software processes for multi-agent systems, including an analysis of their strengths and shortcomings when used in real-world software engineering development.

Based on this, a number of necessary features for a practical, usable software process could be identified. These were used to formulate a number of goals to be met by the newly developed software process.

A new software engineering process, NUMAP (the Newcastle University Multi-Agent Process), was then developed, allowing agents to be created in a structured, repeatable and clearly documented fashion, with a clear series of steps to be followed by the software developer. For such a process to be useful, it needed a high degree of applicability to currently popular multi-agent system development environments.
Additionally, the need for a support tool was identified as a priority, in order to ensure that software developers can take full advantage of the development process, and follow it correctly.

Parts of this work have already been published, primarily related to the development of the software engineering process itself [4], classification of multi-agent systems (Appendix A) [5-7], and on potential applications areas in engineering asset management [8, 9].

1.2 Definitions

While multi-agent systems will be further defined in Chapter 2, it is useful to establish a number of definitions up-front.

An “agent” is defined by the Concise Oxford Dictionary [10] as:

“a person that provides a particular service, typically one organising transactions between two other parties”

Like human agents, software agents act as a service provider, performing tasks independently according to a well-defined role. They do so as autonomous software components, independently performing tasks and using the services of other agents to fulfil their intended purpose.

One particular design paradigm for software agents has been particularly popular over the past decade. This is the “Belief, Desire, Intention” model [11]. This model is based upon “human action”, a discipline of philosophy dedicated to the study of the reasoning behind actions.

This makes an especially compelling platform for structuring software, as the software may be designed in such a way that it mimics the way humans conduct reasoning. Just as object-oriented software engineering introduced a leap of abstraction by introducing the notion of structuring software around elements found in the environment being modelled, BDI multi-agent systems introduce a
new level of abstraction, with the concept of designing software around the way that people reason.

A wide variety of “agent based” systems have been proposed and developed, with varying characteristics in regards to proactive behaviour, adaptability, mobility, collaboration, veracity and disposition. As part of this research, a comprehensive classification scheme for agents has been created, to provide a sense of context to the many and varied kinds of multiagent system. This work has been published [5, 7], and is presented in Appendix A.

1.3 A Process or a Methodology?

The title of this research is “A Software Development Process for BDI Multiagent Systems”. Much consideration was given as to whether to use the term “process” or “methodology” when referring to the design developed herein.

While “methodology” seems to be the more widely used term within the existing literature for systematic approaches to developing software, this was felt to be an ambiguous term, referring to high-level principles and practices, as much as to a discrete series of tasks.

Therefore, the term “process” was chosen to refer to the systematic approach developed from this work, a naming convention in the spirit of the IBM Rational Unified Process [12]. This terminology reinforces the nature of the intended process. It is to be a detailed, prescriptive model for practical software development, not a high-level theoretical framework.

Where alternative processes are discussed, they are generally referred to with their preferred terminology. If the designers refer to a particular process as a methodology, that terminology is used when discussing it.
1.4 Motivation

The impetus for creating a new process for the design of multiagent systems was initially a personal one – a desire to develop software systems using a multiagent framework, but a frustration with the limitations of existing processes available for doing so.

Upon beginning research into agents, the concept of “Belief-Desire-Intention” agents [11] appeared to be an attractive and extremely promising method for designing software.

Software development, over time, has progressed from occurring at machine level – directly programming in the instruction set of the processor architecture, through to more human-readable imperative and structured programming, to object oriented models, where the design of the software is thought about in terms of real-world concepts.

When programming, there is a tendency to describe the software in terms of it having its own reasoning capability. Programmers will often make statements like: “What data does the program need to know to achieve its goal?”, “What state does this module want to maintain?”, “What is the goal of this function?”.

The notion of using high-level concepts, inspired by the notions of human mental reasoning for the specification of software, therefore seems like a natural progression in software abstraction. The BDI model is a solid method for doing so, with practical tools existing for software development.

However, developing software according to this new model with the same design techniques as used for object oriented software development did not seem to be a sensible approach. Many of the advantages of BDI agent design lie in allowing the developer to consider the system at a higher level of abstraction. Using design techniques intended for object-oriented software would
encourage the conceptualisation of the system to occur at a lower level, obviating one of the principal advantages of the BDI approach.

It was clear that an agent-specific process was required for effectively designing multiagent systems. Investigating existing processes showed a large proportion of them were extended from, or heavily based upon, object-oriented design paradigms (e.g. [13-15])\(^1\). While they supported some concepts used in agent design, their origins in object-oriented software development were evident, leading to a development focus that was very similar to object oriented design, rather than one that truly focused on a higher level of abstraction.

Other processes (e.g. [16, 17] [18]) were based on the field of knowledge engineering. While this approach is closer to that used by BDI multiagent systems, it has its own set of shortcomings, taking a centralised view of knowledge in the system, and not focusing on the social, reflective or goal-oriented perspectives inherent in BDI multiagent systems [19].

It was clear a dedicated process was required for developing BDI multiagent systems that was designed from the ground-up to deliver on the promises offered by this model of software development. A number of processes (e.g. [20-22]) have been designed in this manner, and they were investigated in the hope that they would be suitable for designing BDI systems.

Each of these processes possess significant advantages over those that are extended from other techniques. However, they all had their own shortfalls that made them less than ideal. In some cases, the processes were too general, never really getting into enough detail to focus on the “nuts and bolts” of designing BDI agents.

---

\(^1\) Existing processes for multiagent systems will be described in further detail in Chapter 2.
Alternatively, some were designed with such a close coupling to a specific implementation platform that they were difficult to adapt to alternative agent runtime software. In other cases, they only covered a portion of the design process, rather than taking an agent-centric focus from requirements right through to implementation².

It was decided that the best option was to create a new process for developing BDI multiagent systems. A process that truly focused on using an appropriate level of abstraction, using agent-related concepts from the earliest stages of the process, right through to implementation.

1.5 Practical Research

The aim of this research is not only to describe a novel design process for developing agent-based software. The intention of this research has been, from its inception, to create a functional process that not only provides a theoretical framework for multi-agent systems development, but also provides a usable tool that can be immediately put into action for developing software.

A software engineering process has, as its sole purpose, the task of guiding software developers in their goal of developing software. A process that does not do this adequately, or does not provide adequate support tools to allow developers to properly follow the steps involved, is incomplete.

As such, this research focused equally on the theoretical and applied aspects of software development. Each aspect of the research fed into the other. In the early stages of development of the process, a rudimentary support tool was created to assist with following the process end-to-end. In the course of doing so, deficiencies in the design of the process itself were discovered, which led to further revisions of the process, and to further refinements of its support tool.

-------------------

² These issues will be discussed in detail throughout the remainder of this thesis.
Without the theoretical foundations for the process, there would be no procedures and tasks to follow, but without the development of its support tool, the design itself would have been lacking in real-world applicability.

By focusing equally on applied and theoretical aspects of software methodologies, a real-world process has been created that is fully functional, and provides a number of novel aspects not seen in other processes. The proof of the pudding is in the eating, and in this case, the applied aspect of the research greatly enhanced its outcomes, producing a product that is both novel and useful.

1.6 Research Methodology

As described above, the primary aim was to develop a software process that delivered true end-to-end development support for BDI multiagent systems. In doing so, a number of tasks needed to be performed.

Firstly, a through analysis of existing multi-agent system processes was conducted. In doing so, concepts that worked well were noted, areas for improvement were identified, and the goals for the new process were formulated.

Existing agent implementation platforms were also investigated and tested. As these platforms would be the eventual target for software development, a thorough understanding of their functionality was required. Examples of functional multiagent systems were also considered, to identify the possibilities for agent development.

Once this was complete, specific goals for the process were enumerated. A successful process would need to meet each of these aims, and they were used throughout its development to shape design decisions.

A process to meet those goals was then developed, and gradually refined through real-world use. This process draws on the best of what existing
processes have to offer, while offering a number of innovations and a close mapping to real-world multiagent software development. In parallel, the supporting infrastructure for the process was developed. A support tool was created to assist developers, greatly assisting in real-world use of the process and compliance with its procedures.

Finally, the process was evaluated against existing design approaches. In order to do so, a new evaluation framework was developed, using existing metrics drawn from the literature to ensure its impartiality and completeness.

1.7 Structure of the Thesis

The remainder of the thesis is structured as follows.

In Chapter 2, the existing literature is reviewed, presenting background on multiagent systems, the development of the belief-desire-intention model, and on existing processes for developing multiagent software. The goals for the development of a new multiagent systems process are enumerated.

In Chapter 3, an overview is provided on tools and techniques for implementing multiagent systems, and examples of such systems are presented.

In Chapter 4, the NUMAP process is presented, and the design decisions made in its creation are presented. Each phase of process is then described in detail.

In Chapter 5, the NUMAP support tool is introduced, and the decisions made during its development are described.

In Chapter 6, a feature-based evaluation technique for multiagent systems is presented. This draws on existing metrics for evaluating multiagent systems, and extends them to provide a comprehensive approach for evaluating the coverage and effectiveness of the process.

In Chapter 7, the thesis is concluded, and future work is presented.
Appendix A presents a classification scheme for multiagent systems. This assists in placing BDI multiagent system into context among the wider field of software agents.

Appendix B provides details of the mapping scheme that is used for transitioning between phases in NUMAP. This is used by the support tool and dependency tracking mechanism.
Chapter 2 Literature Review

2.1 Introduction

This chapter provides the background and context for the field of multiagent systems, and context of how this topic fits into the discipline of software engineering. The major historical developments that led to the creation of multiagent systems, particularly BDI systems, are identified. Additionally, several prominent design processes for creating multiagent systems are described in detail.

2.2 Software Engineering Background

One of the major promises of software agents is the potential for providing a new level of abstraction for software development, using the unique features of agents to provide a novel method for thinking about software systems.

For as long as software has been developed, designers have sought methods of organising the logic and structure of their software to enhance the usability, maintainability and functionality of the systems they develop. The history of software engineering consists of the development of new techniques and methodologies for creating software, each presenting new paradigms for the conceptualisation, structure and modelling of software systems.

This section provides background on how software engineering techniques have advanced over time, leading to ever-greater levels of abstraction and encapsulation.
2.2.1 Classical Systems Analysis

From the earliest days of creating computer information systems, in the 1950s and 1960s, systems analysis techniques were used to identify the best approaches for constructing software systems. Identifying the data and procedures required is a natural part of software development.

The increased adoption of structured programming [23], and the more rigorous approach to software development that went along with it, led to the formulation of more formal approaches to decomposing systems.

A number of different approaches emerged, such as Structured Analysis [24] and Structured Design [25]. These provided mechanisms for decomposing the system into data and control elements, using techniques such as HIPO Hierarchy Charts.

During the 1970s, the technique of defining a system in structured terms, separating it into individual process objects and data objects, was established. Techniques such as structure charts, data flow diagrams and data model diagrams became the standard conventions for systems analysis and design [26].

Formalisms such as IDEF0 [27] placed an emphasis on decomposing a system into its constituent functions, and identifying data flows between these functions.

These classical systems analysis approaches provided essential tools and techniques for developing complex software systems, and provided the foundation for what was to follow.

2.2.2 Information Hiding

In conjunction with these advances in conventional systems analysis, a number of techniques were developed for segregating design decisions and information
within computer systems. These allow for more complex software to be developed and maintained, by reducing the impact of modifications to one part of the system on the system as a whole.

A critical early development in this regard was the concept of information hiding. First documented by Parnas in 1972 [28], this involves the decomposition of the system into modules that encompass the more difficult design decisions, and hiding the data related to those design decisions to minimise the impact on the rest of the system in the case where those decisions are altered.

Prior to the proposal of information hiding, software was generally divided into discrete sections based upon the major steps in the development process. Parnas challenged this approach, instead proposing that software be decomposed in a way that made future changes and upgrades have less impact on the system as a whole.

This focus on composing software systems from discrete modules led to the development of the MODULA-2 programming language by Wirth [29]. Growing out of PASCAL, it provided syntax and structures that allowed modules to be explicitly defined, and separately compiled. MODULA-2 provided further mechanisms for hiding internal logic and design decisions, by separating modules into “definition modules” that are visible to other modules, and “implementation modules” containing internal code that is to be hidden from other modules. This was achieved by strict scoping. Only those parts of a module that are exported are visible at all to other modules.

The development of modularisation and information hiding provided a much-needed basis for the construction of complex software systems that could be readily modified and upgraded, and also allowed for better reuse of individual modules due to the resulting decoupling of data structures.
2.2.3 Object Oriented Software Engineering

The development of objects as a concept for specifying software systems also offered advantages in the development of complex systems. Formally introduced in the programming language, Simula 67 [30], and based on Hoare's concept of record classes [31], objects provide a means for developing software by modelling components based on analogous real-world or conceptual “objects”. This offers an alternative way of structuring software with a focus on information hiding, by grouping the information and logic of a software component together into an individual object.

The introduction of Smalltalk-80 [32, 33] by Alan Kay's research team at Xerox PARC provided an extension to this concept. All data defined within Smalltalk-80 is represented by objects (there is no primitive typing), and all interaction between objects is done via messaging. Objects are defined by classes, and do three things: hold private information representing the state of the object, receive messages, and pass messages.

Object oriented programming is based upon three main principles: encapsulation, the keeping of state information about the object private; inheritance, by which subtypes of a given object inherit attributes and behaviour from their supertypes; and subtype polymorphism, by which subtypes of a given class can be invoked in-place of their supertypes.

Object orientation allows software to be developed in a largely decoupled fashion, offering many of the same advantages as modularisation and information hiding, but with a different conceptual framework. Designing software using an object oriented approach means thinking about the physical and virtual objects that exist within the scope of the system, and designing individual software elements to represent them.

This approach was further enhanced by the adoption of use cases as the main driver for software design in the “Object Oriented Software Engineering
Methodology” [34], which later evolved into the Rational Unified Process [12], one of the cornerstones of modern software engineering practice.

2.2.4 Component-Based Software Engineering

As software systems have grown larger, additional high-level concepts for grouping software into discrete modules have been developed. Decoupling software modules can be done on a grander scale, grouping together collections of objects into modules of their own. An early advocate of this approach was Cox, who proposed the concept of Software ICs (developed originally with Objective-C) [35]. This approach is based around developing a library of components that can be integrated and reused to create software systems. It is now better known as component-based software engineering.

2.3 Multiagent Systems

All of the above advancements are based on developing new ways to think about software development, with a focus on developing more complex systems while retaining a strong degree of maintainability and upgradeability. The successful advances highlighted above required both a new paradigm for creating software, and adequate tool or language support for doing so.

Multiagent software engineering, in particular BDI multiagent systems, can be seen as another step in this evolution. Multiagent systems present a promising approach that further advances the field of software engineering by providing a new paradigm for developing software.

This idea is supported by Wooldridge [20], who proposes that software development is advanced by using additional levels of abstraction. "Procedural abstraction, abstract data types, and, most recently, objects and components are all examples of such abstractions". Multiagent systems provide the next advance in this regard.
2.3.1 Overview of Multiagent Systems

The field of multiagent software is a broad and rapidly developing area of research, which encompasses a diverse range of topics and interests. The rapid growth of this field in the past decade has occurred in parallel with the evolution of the World Wide Web, because multiagent systems show great promise for exploiting massively distributed systems, including those that are Internet-connected.

Despite the popularity of multiagent systems, there is very little consensus about what exactly constitutes a software agent. Research into agent-based systems has been quite diverse [36], which makes it difficult to form a comprehensive definition for a software agent.3

While there is a large range of different approaches for agent design, several key features are common to all multiagent systems. The fundamental feature of software agents is autonomy. Software agents must be able to perform their intended functions independently. Agents must be able to take action when necessary without human interaction or confirmation.

As a result of their autonomy, software agents must run continuously. Unlike much conventional software, which performs a fixed task then terminates, agents must run in a continuous “loop”. This allows agents to monitor the current situation and take appropriate action when required. Agents also possess social ability, that is, the ability to interact with other agents. The real advantages of software agents come not from individual agents, but from communities of interacting agents.

Software agents are situated within some form of environment. This may be, for example, a simulation, a single computer, a large-scale distributed network such

3 See Appendix A for a classification scheme that places the various streams of multiagent systems into some context.
as the Internet, or a real-world, physical environment. Information about the environment is received by sensors, the environment may be altered by effectors, and messages may be received from other agents.

2.3.2 Definition

In essence, a multi-agent system can be defined as: a system consisting of autonomous, proactive software components that interact in order to achieve goals by utilising their own individual knowledge about a complex, changing environment.

2.3.3 Characteristics of Multiagent Systems

In the same way as objects can be identified by a set of attributes in common to their design (namely message passing, inheritance, polymorphism and encapsulation), software agents have a number of core features, along with some optional characteristics. In order to fully understand multiagent systems, these characteristics need to be identified.

A number of authors have attempted to identify the key characteristics that define software agents. For example, a typology of agents has been presented by Nwana [36], who identifies an “agent” as a meta-term, covering a range of agent types. The three primary attributes that agents should exhibit are identified as autonomy, learning and cooperation (however these are not proposed as being necessary for a software component to be an agent). Further, two kinds of distinctions between agents are also defined, static versus mobile and deliberative versus reactive. The first distinction refers to the ability of agents to move around a network, while the second refers to whether they engage in planning and reasoning based upon an internal symbolic reasoning model.

Another classification is presented by Franklin and Graesser [37]. In this classification all agents are considered to be reactive, autonomous, goal-
oriented and temporally continuous, with the following attributes being optional: communication, learning, mobility, flexibility, and character (personality).

Wooldridge and Jennings proposed definitions of properties for software agents [38]. A weak notion of agency is proposed, defining an agent as a hardware or software system with the following properties: autonomy, social ability, reactivity and pro-activeness. One notable aspect of this definition is that it requires an agent to both react to its environment, and exhibit goal-directed behaviour, i.e. to be both reactive and pro-active. This is in contrast to several other classifications listed above, which consider purely reactive and purely goal-oriented systems within their classifications.

A stronger and more specific notion of agency is also identified by Wooldridge and Jennings. In addition to the properties above, agents adhering to this strong notion of agency have some additional characteristics. These include: mobility, veracity, benevolence, and rationality. It may be noted that “adaptiveness”, or learning, is not explicitly mentioned within this classification at all.

Based on these existing classification schemes, the core attributes of software agents may be identified as autonomy, temporal continuity, and social ability.

Agents must be able to run independently, with little or no human intervention, therefore autonomy is a necessary property of agency. Temporal continuity is required, as agents must run continuously, rather than simply perform a task, and terminate. Agents must also possess some form of social ability. The real advantages of software agents come not from individual pieces of software acting in isolation, but from communities of interacting agents.
In addition to the core attributes, agents may be classified according to the following features:

- Pro-activeness
- Adaptiveness
- Mobility
- Collaboration
- Veracity
- Disposition

A comprehensive classification scheme based upon these features has been created as part of this research project [5, 6], and is described in detail in Appendix A.

**2.3.4 Advantages of Multiagent Systems**

Multiagent systems present a number of advantages over previous software development techniques.

A brief overview of these advantages is provided in this section. More detail regarding the advantages of multiagent systems will be provided in Chapter 3, where the specific advantages of Belief-Desire-Intention multiagent systems are presented.

Multiagent systems provide a high degree of encapsulation and abstraction. Systems are necessarily divided into a number of individual, autonomous components, each fulfilling a number of specific goals. The design process for each agent can focus on fulfilling those goals, without needing to focus on the system as a whole. This provides a high degree of encapsulation. A separate design phase, concerned with integrating the specific agents into a coherent system can then be performed. Agents are, in essence, strongly-encapsulated software components. It has been argued that this encapsulation allows for enhanced maintenance and reusability [36].

Agent-oriented design provides an ideal method for designing concurrent systems. Each agent is its own autonomously executing component, and the
interactions between agents are performed, in systems such as Jadex[39], by asynchronous message passing. This means that agents can be easily implemented with separate threads of execution, providing an inherent facility for concurrency within the system.

In addition to their concurrency, multi-agent systems provide a means of *distributed control* over a system. Rather than having one central thread controlling the flow of execution, agent-based systems allow for control to be distributed among all the entities in the system, each pursuing its own individual goals. This autonomy allows agents to be geographically distributed, executing on independent agent platforms.

External systems may be easily integrated into multi-agent systems, with an interface agent acting as a “wrapper”, allowing for simple *encapsulation of legacy systems* [40].

In situations where graceful *recovery from failure* and response to real-time perturbations is required, agents may offer a suitable approach [41]. If an agent fails within the system, other agents using its services may seek out alternative service providers, offering a mechanism for the system to recover.

These advantages make multiagent systems a promising development technique, particularly for software that is embedded in complex environments.

### 2.3.5 Challenges in Developing Multiagent Systems

Due to the unique nature of software agents, there is a number of distinct challenges faced by designers of multiagent systems.

*Well-defined interaction protocols:* Communication between software agents is asynchronous in nature, and is performed in a best-effort manner. Interactions therefore need to be designed in a robust manner, handling failure conditions, and allowing for recovery if requests to other agents are not handled in a timely manner. Well-defined interaction protocols are needed in all aspects of agent
communication, and focus needs to be placed on the “social” design of the system early in the development process.

*Goal-oriented design:* The actual internal design of software agents varies depending on the type of agent used. The most popular, and promising, approach to developing multiagent systems defines agents in terms of the agent’s belief, desires and intentions. This poses challenges for the design of agents, as it is an approach that differs significantly from conventional software development. The key to properly designing this kind of software agent is to correctly specify the desires (or goals) to be achieved by the agent. This *needs to be a focus from the early design stages, right through to detailed agent design.*

*Platform support:* A number of different agent platforms are available for implementing and deploying agents, with each of these platforms undergoing rapid development. Therefore, care needs to be taken in designing multiagent systems to not tie the design too closely to the specifics of a particular agent platform. Additionally, the lack of detailed documentation and IDEs for some agent platforms means that more care needs to be taken during implementation, as development can be very error-prone.

*Dependency management:* One of the largest challenges faced in designing multiagent systems is managing the complex web of dependencies that may exist in the system, both between agents and internally. While agents themselves can be considered autonomous, the system as a whole relies on the complex interactions between agents to provide its overall functionality. Analysing the dependency relationships between agents is necessary to highlight how changes to the functionality of one agent may affect the rest of the system, and to determine how failure within a single agent may affect the rest of the system.
2.3.6 Need for a Process

These challenges can be partly addressed by using a well-designed, appropriate process for developing multiagent systems. Existing processes for software development, focusing on other approaches (such as object-oriented design) are inadequate, as they do not address the specific challenges faced within multiagent systems, and they do not exploit the additional levels of abstraction offered by multiagent systems.

A successful process for multiagent system development needs to take into account the specific properties of multiagent systems, such as the goal-based nature of the design process and the organisational complexities between agents. The process needs to balance the need to properly specify agents in terms of the conceptual framework used (e.g. using beliefs, desires and intentions), but also maintain flexibility to cater for new and updated agent environments.

These requirements make it clear that specific design processes are needed for multiagent systems, designed from the ground-up to cater for the advantages and challenges faced in developing such systems.

2.3.7 Software Processes are Software Too

A software process is not simply an arbitrary series of steps to follow in developing software, it is a systematic approach to be used to improve the quality of the software. Software processes describe a way of going about the work of software development in an orderly manner, in order to best achieve an outcome that delivers a quality software product.

Osterweill famously noted that “Software processes are software too” [42]. Just as with software, processes are described in detail, then instantiated and executed. A process description, like an object-oriented class, describes what is to be done, and each project in which that process description is used is akin to
an instantiation of that class as an object. In designing process descriptions, it was suggested by Osterweill that “contemporary programming techniques and formalisms be used”. The concept of “process programming” applies the best practices in software development to the design of software processes.

In designing a software process, or more formally, a “process description”, inspiration should be taken from the techniques used in software development. Not only in the design and structure of the process itself, but also in the use of software tools to directly “execute” the process, supervising and supporting the efforts of software developers.

Osterweill notes that this perspective should also extend to the design of software processes. In developing software, we should engage in a clear requirements analysis, to identify the goals of the process in terms used in the analysis of software – such as robustness and functionality [43].

This approach has been used in the design of NUMAP. The task of creating a process for multiagent systems was seen as being akin to developing a robust piece of software. Good software processes, like good software, must be flexible, adaptable, robust and complete.

### 2.3.8 Strategies for Process Design

In designing a software process, consideration needs to be given to the general approach taken to grouping and organising development activities. In current Software Engineering practice, two contrasting approaches dominate discussion: the traditional *Waterfall* model [44], and *Agile* processes [45-47].

The *Waterfall* model, originally derived from the work of Winston W. Royce [44], is an approach where software development proceeds in order from *requirements specification*, to *design, implementation, testing* and *maintenance*. Each of these *phases* is verified before proceeding to the next.
This model is often criticised due to its rigid nature [48], and it represents an ideal structure rather than a realistic process, as it is unlikely that any software project of significant complexity will proceed from requirements specification to testing without needing to backtrack and make changes to the output of earlier phases. Due to this, the unmodified Waterfall model is rarely used in practice. Indeed, Royce later recommended that feedback should lead to revision of earlier stages of the process [49], as results from testing lead to design changes, and design constraints lead to requirements being revisited.

In real-world software development, phased approaches rarely, in fact, follow the top-down waterfall approach. Instead, they follow an approach based on iterative and incremental development. This uses the same phased approach as the waterfall model, but with individual phases being repeated as many times as required to generate a product that is fit for deployment. This “evolutionary” approach to software development is used in popular design processes, such as the Rational Unified Process [50].

In iterative approaches, the speed of iteration through cycles of design can vary between processes, and change based on the needs of individual projects. Some projects may require rigorous design up-front, while others may benefit from rapid development of a prototype.

A class of software development approaches, referred to as Agile methods [45-47] place a particular focus on very rapid incremental development. These approaches use short development cycles, and de-emphasise formal development phases as the main units for segmenting software development.

In the case of the Scrum [51] framework, the focus is on designating well-defined roles within a team, and then performing extremely short development iterations, called “sprints”, with well-defined development targets. These each have a fixed duration, with the expectation of having functioning output at the end of the iteration.
Another popular Agile methodology, *Extreme Programming (XP)* [52], similarly places a focus on short development cycles. It de-emphasises documentation, and places coding and testing at the centre of development. A number of practices and rules are adopted during development, including: “pair programming”, where programmers work in pairs on a single workstation; and “test-driven development (including unit testing and frequent integration testing). Another focus of XP is communication through institutional knowledge, rather than primarily through documentation.

A focus of Agile development is on avoiding “big design up front” [53], instead focusing on first developing “the simplest thing that could possibly work” [54]. This has been criticised, particularly by software development commentator Joel Spolsky, who has stated that avoiding “thinking things out in advance” can lead to “serious development headaches” as a project proceeds [55].

Agile approaches provide benefits from a project management standpoint, benefiting from the rapid completion of deliverables. However, the de-emphasis of documentation can be detrimental when developing large software systems with complex interfaces. In the case of multi-agent systems, where well-defined interfaces between components are vital for ensuring consistent communications between agents, this is of particular concern. Rapid iterations and team-based development may well be appropriate from a project management standpoint, but this should not be at the expense of documentation for interfaces and communication ontologies in multi-agent systems, where these are critical to a system’s success.

### 2.4 Historical Development of Multiagent Systems

This section provides an overview of the historical development of multiagent systems, from early agent-like systems, through to modern systems using the Belief-Desire-Intention paradigm.
2.4.1 Early Agent-Based Approaches

A number of historical approaches have been taken to developing systems that behave autonomously, with or without abstract reasoning.

2.4.1.1 Subsumption Architecture

One such early approach was the Subsumption Architecture, designed by Rodney Brooks [56]. This architecture provides a reactive framework for physical robots, which uses layers of reactive control systems to achieve complex behavior.

The Subsumption Architecture allows the designer to add additional "layers of competence" to an agent over time. While originally designed for physical robots, it has since been adapted for software agents [57]. The agent designer can hence expand and adapt the agent’s functionality over successive iterations of development by introducing new reactive layers. This differs from agent learning, as the adaptation is performed by designers explicitly adding functionality to the agent.

The architecture is based upon the concept of building layers of control to allow the robot to operate at increasing levels of competence. Higher level layers can subsume (encompass) lower levels by suppressing their outputs, although lower levels continue to operate.

Each higher level can observe the lower level, and inject data into its internal interfaces to suppress normal data flow, however, the lower level does not explicitly know about the higher level. This allows the control system to be responsive to high priority goals, while still servicing necessary low-level goals. Complex behaviour emerges from interaction with a complex environment, rather than from an inherently complex system.

Individual layers can be working on individual goals concurrently, with the suppression mechanism mediating the actions that are taken. If a higher level
does not produce results in a timely fashion, the lower level will continue functioning at a lower level of competence. Since lower levels have been extensively debugged and tested, this results in a robust system.

The Subsumption Architecture pioneered several concepts that are now taken for granted in the field of agent design, particularly the concept of autonomous processing without the need for traditional AI techniques.

Brooks famously stated that “elephants don’t play chess” [58], meaning that symbolic logic and reasoning are not required for complex behaviours. Appropriate, sufficiently complex behaviours can emerge spontaneously from coexistence and cooperation between physically grounded systems. This notion of systems exhibiting complex behaviour via interaction with each other and their environment is a key principle of modern multiagent systems. Etzioni has stated that “intelligent agents are 99% computer science and 1% AI” [59].

While the concept of “layered” levels of control, with suppression of lower levels, is unique to the Subsumption Architecture, parallels can be drawn between this approach, and modern software agents that may be classified as having “hybrid” proactivity [5].

Despite pioneering these now-common concepts, the Subsumption Architecture itself has been relegated to being used primarily for robot design. The complexity of having many layers of functionality, the lack of runtime adaptability, and the difficulty in managing the processes of suppression between layers are all key disadvantages of the model.

2.4.1.2 Swarm-Based Agents

Another early agent design approach involved “swarm intelligence”. These systems are comprised of a large number of independent agents, often of low complexity, interacting with their environment and each other. These agents are often used to simulate self-organising systems.
A key implementation with these features is IBM’s SWARM [60]. This provides a framework within which communities of agents can interact. SWARM has seen widespread adoption for simulating social systems. The SWARM agents themselves can vary in complexity to a large degree. For example, the HeatBugs [61] model demonstrates emergent behaviour using very simple rules, and without using direct communication between agents.

SWARM was an early pioneer in demonstrating the potential of self-organising systems, but is largely limited to this purpose. Its simplicity restricts its use for more complex applications.

### 2.4.1.3 User-Centric Agents

There has been considerable research into “user-centric agents”. These are agents that are designed to assist users with a particular task. Such agents are driven by their interaction with users, rather than other software. This interaction generally involves the provision of specific information to a user. The information may be specifically entered by the user, drawn from a knowledge-base, or generated by learning behaviour [62].

An example of such agents has been proposed by Maes [62]. These agents provide a user-interface that assists the user by acting as a "personal assistant". Rather than the user directly manipulating information, these agents use "indirect management". The user is engaged in a co-operative process with the agent, whereby the agent adapts to the preferences of the user by learning from the user’s behaviours. This approach was applied to email, scheduling, news selection and music selection domains.

A similar approach was applied to the domain of organizational meeting scheduling by Sen and Durfee [63]. These agents exist for each user and exchange information to “intelligently” schedule meetings within certain constraints. The adaptiveness of this approach lies in the selection of meeting
scheduling algorithm. By adapting to individual users, the right algorithm can be chosen for the right situation.

However, the learning processes involved in these “personal agent” systems rely on repetition of actions by the user, with the agent adapting based on the user’s selections, allowing the agent to subsequently suggest options based on past selections. The approach would be limited to applications in which such repetition occurs, and in which individual users’ behaviours differ significantly.

2.4.1.4 Analysis of early approaches

Early approaches to agent design demonstrated the promise of using adaptive, autonomous components for developing software. They introduced the notion that systems can demonstrate complex behaviours through the interactions of seemingly simple rules, and can adapt to their inputs to improve the performance of the system over time.

However, the early approaches were developed with a variety of different techniques, sharing very few key features other than autonomy and adaptive behaviour. A more coherent framework for developing software agents was needed.

2.4.2 Development of the Belief-Desire-Intention (BDI) model

The Belief-Desire-Intention (BDI) model has proven itself as the most widely adopted approach for developing multi-agent systems [64], with a number of mature, robust software implementation platforms based upon the paradigm [39, 65].

The BDI approach provides a framework for autonomous cooperation and reasoning within multi-agent systems, defining how agents behave at an internal level. This is achieved using theories developed from the philosophies of human reasoning, specifically the belief-desire-intention model of human action.
2.4.2.1 Philosophy of human action

The origins of BDI agents lie in the philosophical theories of “human action”, which aim to describe the reasoning behind actions.

One of the traditional theories of human action, the desire-belief model, states that actions are motivated by a combination of the desires and beliefs of the actor. A new belief-desire-intention model for human action was created by Bratman in the 1980s [3], as a rejection the desire-belief model. This new model asserts that intentions are a distinct phenomena from the desires and beliefs that motivate those intentions. Desires, beliefs and intentions are considered the basic elements underlying intentional action.

Bratman’s model includes two kinds of intention: future directed intentions and present-directed intentions. Future-directed intentions are intentions to perform some action some time in the future. These intentions act as elements in coordinating an entity’s plans, constraining other intentions in order to achieve consistency. Present-directed intentions guide behaviour, triggering actions that attempt to achieve the intended effect.

2.4.2.2 BDI and Means-End Reasoning

Bratman, Israel and Pollack applied these theories to the field of Artificial Intelligence (AI) [1]. In AI, the “planning problem” addresses the issue of means-end reasoning, i.e. the process of reasoning about the steps required to achieve a goal. The authors stated it was important to note that the time taken to select a plan is limited, as agents are resource bounded – there are constraints on how much processing they can do in a given interval of time. Therefore, the time taken for means-end reasoning is an important factor, as many tasks are time-limited, and the conditions for selecting a plan may change while the agent takes time deliberating about plan selection.
The authors proposed a BDI architecture that facilitates means-end reasoning and selection of alternative courses of action within a resource-bounded context. The notions of belief, desire and intention are explicitly used in this architecture. Intentions are represented as elements of larger plans, and desires and beliefs are represented as data stores.

The architecture provides an overall conceptual model for planning. “Plans as recipes” are stored in a plan library, and represent beliefs about the actions that should be taken when specific conditions are met. In addition to traditional means-end-reasoning, the architecture defines a filtering process that allows plans that are inconsistent with the agents’ beliefs and existing plans to be ignored. An opportunity analyser allows options to be proposed in response to perceived environmental changes. A filter override mechanism allows options that do not survive filtering to be considered if they meet certain override conditions.

This model also includes the idea that plans may be partial. There is a need for partial plans in changing environments, as highly detailed plans about future events are likely to become outdated. Two forms of partial plans are proposed: temporal partiality and structural partiality. Temporal partiality refers to plans that do not specify events to be undertaken for specific periods of time, and structural partiality refers to plans that may specify ends rather than means, leaving the deliberation on how to reach these ends for later. Structural partiality is a key part of modern BDI systems. Plans may specify that a goal needs to be achieved, but leave the deliberation on how to achieve that goal until it is necessary. Temporal partiality is also a key aspect of agent systems. In changing environments, it is crucial to have the ability to specify that a goal needs to be achieved in the future, but not specify the exact mechanism by which it will be achieved until later. This addresses the problem that the state of the environment at a later point in time cannot be known, and reduces the amount of means-end reasoning that is required at any one time.
The BDI architecture provides an important foundation for agent systems, but does not specify the precise mechanics by which the actions such as filtering and opportunity analysis are to be undertaken. A more formal approach was required to prove the validity of the approach.

2.4.2.3 Formalisms for the BDI architecture

Cohen and Levesque

The first such attempt at providing a formalism for the BDI architecture was presented by Cohen and Levesque [66]. This formalism is aimed at defining the key concepts used in BDI systems, namely beliefs, goals, plans, commitments and intentions.

In developing the formalism, they defined relationships between various concepts within BDI agent systems. Goals are defined as being desires that an agent has chosen to pursue, and intentions may be modelled as a kind of persistent goal. As a result, intentions are essentially reducible to being a kind of goal, which is itself a kind of desire.

Several points regarding intentions within BDI systems were also raised. The authors note that plans are not intentions, as intentions indicate a course of action that an agent has chosen to pursue, while agents may never decide to adopt a given plan. Agents should commit to intentions, discharge intentions that are believed to have been satisfied, and adopt new intentions when plans change.

Intentions that an agent is currently acting upon are called present-directed, while intentions that the agent intends to pursue subsequently are referred to as future-directed. Present-directed intentions control the behaviour of the agent, while future-directed intentions constrain the adoption of other intentions – ensuring that potential intentions do not conflict with adopted ones.
A key issue is apparent in this formalism. The reducibility of intentions to goals, which are themselves a form of desire, goes against the core principle of the BDI model – the primacy of intentions. Therefore, Cohen and Levesque’s formalism for BDI essentially reduces the belief-desire-intention model to the belief-desire model that preceded it.

*Rao and Georgeff*

In an alternative formalism, Rao and Georgeff [11] criticise the reducibility of intentions to other concepts. They instead define a possible-worlds formalism where intentions are “first-class citizens on par with beliefs and goals”. In their formalism, intentions are considered to be partial plans of action that an agent is committed to execute in order to fulfil its goals.

This formalism models the world as a temporal structure with a branching-time future and a single past. This structure is called a *time-tree*. Branches are choices available to an agent at any one time. There is a set of belief accessible worlds – there can be multiple worlds due to the agent’s uncertainty about its environment. Additionally, there is a set of goal accessible worlds, representing the goals of an agent. The difference between goals and desires is that goals must be consistent with one another, whereas desires have the possibility of not being consistent.

A set of intention accessible worlds is also modelled. These are the worlds that the agent has committed to realise. The set of intention-accessible worlds must be compatible with the set of goal accessible worlds, as an agent can only intend a course of action if it believes the action will satisfy one of its goals. Goal-accessible worlds and intention-accessible worlds are selective choices from the belief-accessible world.

This formalism provides a superior approach to that described by Cohen and Levesque. The primacy of intentions is one of the key aspects of Bratman’s
model, and this formalism is based upon intentions. It provides a suitable model for BDI agents, and a foundation upon which actual BDI systems can be based.

### 2.4.2.4 BDI in practice

With the BDI architecture formally defined, development of multi-agent systems based on this approach commenced. The first practical software system to be based on BDI principles was the Procedural Reasoning System (PRS) [67], developed by Georgeff and Ingrand.

This system proved to be enormously influential, and set the groundwork for future BDI software systems. The Procedural Reasoning system and other BDI platforms will be further discussed in Chapter 3.

### 2.5 Software Processes for Developing BDI Multiagent Systems

With the background of multiagent systems and the BDI model established, the next focus is on processes for developing multiagent systems. These will be presented in this section.

#### 2.5.1 Aims for a Multi-Agent Software Processes

In developing a software process, or methodology, for BDI multi-agent systems, the distinct features of such systems need to be taken into account. A process is needed that is specifically designed to cater for the challenges and opportunities faced in multi-agent software development.

In particular, a key aspect of BDI agents is their basis in Bratman’s theories of “human action”. Such a software process needs to treat beliefs, desires and intentions as the principal elements used in design. If these concepts are not used in the design of a multi-agent system, then the agents in question can hardly be seen as being “BDI” at all.
Additionally, the software process needs to allow for defining the organisation, collaboration and communication between agents within the system. As agents themselves are largely autonomous software components, the process of creating a coherent software system from them is largely a process of defining their interactions.

Finally, a software process must have full life-cycle coverage, from initial requirements specification, all the way through to detailed agent design and implementation. Without such specificity, the process would be very difficult for software practitioners to follow.

An overview of a number of existing software processes for developing multi-agent systems follows.

**2.5.2 Gaia**

The Gaia methodology [20], developed by Wooldridge et al details a method for developing agent-oriented software. Gaia is a methodology for defining both the macro-level (societal) and micro-level (individual agent) aspects of agent-based systems. The design is based on a collection of agents consisting of various different roles.

Gaia is divided into two distinct phases, analysis and design. Analysis is primarily concerned with understanding the system to be developed, rather than with design details. Roles and relationships are defined, representing the system’s organisational structure. The system as a whole is seen as a single organisation.

At the design phase, roles are aggregated into agent types, and services are specified for the agents, representing the tasks they are able to perform for other agents in the system. Communication pathways are identified through an “acquaintance model”.
Gaia’s primary limitation is that the design is extremely abstract. Interactions are defined loosely, with the purpose of the interaction being specified, but not the series of steps that need to be taken. Agents are also defined at a high-level, with the implementation details being omitted.

The aim of having such an abstract design phase was to allow the developer to “transform the analysis models into a sufficiently low level of abstraction that traditional design techniques (including object-oriented techniques) may be applied”. Detailed design is considered to be “beyond the scope of Gaia” [20].

While having an abstract design allows the model to be applied to a wider range of implementation platforms, it seriously detracts from the ability to implement the design. A comprehensive design process that maps closely to implementation has the potential to reduce ambiguities in implementation and ease the development process.

In particular, GAIA’s sole focus on the late “analysis” and high-level “design” phases of development limits its effectiveness as a real-world agent development process [68], especially considering the present availability of quite advanced tools for directly implementing BDI multiagent systems.

An extension to the GAIA methodology does exist for mapping the process specifically to the JADE [69] implementation environment, called GAIA2JADE [70]. However, this methodology specifically constrains itself to the base JADE framework, which is not oriented around the BDI model, and has limited applicability to other implementation environments.

2.5.3 Tropos

The Tropos methodology [21] is based on two key ideas. Firstly, the concept of agent is used throughout the development process, from early analysis to implementation. Secondly, unlike most agent development methodologies, Tropos deals with the early requirements analysis process, with the goal of
attaining a better understanding of how the environment, agents and users interact. Tropos aims to have the entire development process take place at the knowledge level, rather than introducing notions from traditional software development.

The authors state that one of the primary uses of agent-oriented development is in developing flexible, changing, open architectures. Programming at a knowledge level allows a software system to evolve and change to meet the need for new requirements and components within changing systems.

There are five main stages to the Tropos methodology: early requirements, late requirements, architectural design, detailed design, and implementation. Early requirements analysis involves determining the stakeholders within the system, and identifying their intentions, determining the goals of each actor, and determining the dependencies between actors. Specification of goals for actors is not a task performed by the design team, but rather a decision for the associated stakeholder.

Within Tropos, there are two kinds of goals. “Hardgoals” are standard goals that may be satisfied by achieving some form of satisfaction criteria. “Softgoals” are not as clearly defined, and do not have a satisfaction criteria. Actions may contribute towards a soft goal, but the goal can never be fully satisfied. As a result, soft goals may be used to model non-functional requirements.

Late requirements analysis defines the system that is to be developed, and specifies how it will interact with its environment. Actors are modelled, and dependencies with other actors in the organisation are defined. Decomposition of goals also takes place in this phase, with means-end analysis, contribution analysis and AND/OR decomposition being used to analyse the goals.

The architectural design phase focuses on the system’s global architecture, defining the subsystems and their dependencies. Subsystems may be connected via data flows and control flows. New actors are added, based upon discoveries
made during the analysis of the architecture, and capabilities for these actors are defined. Based on the actors that have been identified, agent types are created and are assigned capabilities.

Detailed design in Tropos involves the design of agents’ “micro-level”. Agent properties are defined, including goals, beliefs, capabilities and communication with other agents. Tropos uses UML activity diagrams for representing capabilities and plans, and uses Agent UML [71] derived diagrams for agent protocols.

While Tropos offers a comprehensive approach for determining the requirements of an agent system, it lacks the ability to specify detail about the internal reasoning of agents. The detailed design phase focuses on using AUML [72] to describe the agent, and provides only limited specification of how agent plans are to be defined, how they are to be selected, and how the agent will react to specific communication messages. It has been observed that Tropos is “not suitable for sophisticated software agents requiring advanced reasoning mechanisms for plans, goals and negotiations” [73].

2.5.4 Prometheus

The Prometheus methodology [22] attempts to cover a large portion of the agent development life-cycle, from specification and definition of the system, through to implementation. The process is based upon real-world experience with developing agent-based software in the JACK environment [65].

Prometheus is divided into three main stages: system specification, architectural design, and Detailed Design. Components of the specification and design, such as forms and diagrams, are termed artefacts. At each stage, a number of artefacts are defined, representing aspects of the system design, and are stored in the form of structured data.
In the Prometheus process, high-level goals for the system are defined at the 
*system specification* stage. This phase involves capturing these goals while 
performing the process of high-level analysis and design, by defining the inputs, 
outputs, external data sources and process-oriented scenarios for the system. 
As Prometheus does not have a dedicated requirements elicitation phase, this 
high level design serves to both capture the requirements of the system and 
specify its high-level functionalities.

The artefacts defined at the system specification level include high-level 
concepts such as scenarios, system goals, functionality descriptors and basic actions, in addition to the system's inputs and outputs.

At the next stage, *architectural design*, a high-level view of the agent types in the 
system along with their interactions and communication protocols, are defined. 
This is done via a bottom-up, functionality-centric process, taking the 
functionality descriptors from the previous phase, and grouping them to define 
the agents within the system. In doing so, care must be taken to ensure the 
creation of cohesive agents. After this task is complete, interaction protocols 
between the agents are defined, along with message descriptors and an overall 
view of the system structure.

At the architectural design stage, artefacts include agent types, along with an overview of the system's structure and a description of interactions between agents.

This is then followed by the *detailed design* stage. State information for agents is defined, identifying their internal processes, then “capabilities” (behaviour descriptors) are defined for these processes. Plans are created to describe the detailed behaviour within capabilities, and events and messages that trigger these plans are identified.
At the detailed design level, agents are defined in terms of their capabilities, plans, and the events they handle. Also, the data that is stored by the agents is defined, although the means of doing so is not specified in detail.

The designers of Prometheus state that designs achieved using the methodology can be implemented in a number of agent environments. However, the sole target implementation environment in examples given in the designers’ book [22], and used for code generation in the supplied support tool [74] is JACK [65]. This specificity to a particular implementation platform, along with the absence of a dedicated requirements elicitation phase, are the main drawbacks of Prometheus.

2.5.5 Agent OPEN

Another design approach for creating multiagent systems is Agent OPEN [75]. This technique extends the OPEN Process Framework (OPF) [76] to support the design of multiagent systems.

The OPEN Process Framework defines a “metamodel” for creating customised methodologies, based on the individual needs of a project or organisation. OPEN is based upon the principles of situational method engineering, allowing tailored methodologies to be created.

OPF provides a repository of process components that may be selected and assembled according to construction guidelines. The construction guidelines assist the user in selecting the method fragments to be used in the customised process.

The framework was originally intended for object-oriented development, however it has been expanded to support agent-based software. This extension is referred to as Agent OPEN [75]. The extended version augments the existing process components with a number of additional work units that were added after examining existing agent methodologies, such as Tropos, Gaia and
Prometheus. [76]. These additional units fall primarily in the “Requirements Engineering” and “Build” phases of development.

Agent OPEN does not provide a standalone methodology by itself, rather it provides a mechanism for constructing methodologies from existing process fragments (derived from a number of multiagent design processes). This is an interesting approach, however, its use requires expertise in “method engineering”. It does not provide a standalone, whole-of-lifecycle process, rather it creates tools to create a new one using method fragments derived from existing processes.

2.5.6 Organisational Approaches

A number of papers on multiagent system design have suggested taking a business process-centric or organisational approach to designing multiagent systems [77-80]. An explicit organisational structure may make such systems easier to design and more stable in operation.

The structure of the organisation may be dynamic or static [78]. Static organisational structures are defined at design-time, and hardcoded into the system, whereas dynamic organisational structures are established at run-time.

A dynamic organisational structure allows agent organisations to be formed as a result of agent relationships established during the system’s execution; the processes for establishing organisations are determined during design, but the actual memberships of organisations are established during the operation of the system.

Partitioning a multiagent system into organisations presents several advantages [80]. Firstly, it leads to modularity, which in turn increases encapsulation within the system. Secondly, organisation-based design can help bridge the conceptual gap between the real-world and the system, allowing real-world organisations to be better represented within the system. Also,
ensuring that agents play well-defined roles within an organisation assists with achieving reliability within the system. Finally, explicitly defining the structure of the organisation helps define how agents will interact.

As such, it is worthwhile investigating organisational approaches to design.

### 2.5.6.1 The Extended GAIA Methodology

One of the more prominent organisation-based processes for developing multiagent systems is presented by Zambonelli et al in their extensions to the Gaia methodology [80]. These extensions refine the Gaia methodology to allow it to better support organisational concepts. The refined methodology is itself, confusingly, referred to as *Gaia*.

The extended Gaia model adds an *architectural design* phase, and placing an emphasis on defining the system’s organisational structure. Additional concepts are modelled, such as the *environment* in which the organisation is situated, *organisational rules* (constraints that are placed on agents within an organisation), and *organisational structures* (defining a topology for interaction patterns between roles and the control structure of the organisation).

Possible organisational structures include a hierarchical structure or a market-based one. Despite allowing dynamic alterations in the membership of organisations at run-time, the interaction patterns and the control structures for the organisation are defined at design-time.

However, like the original Gaia methodology, “extended Gaia” does not directly deal with modelling agent internals, nor does it consider implementation issues or the requirements elicitation process. As such, its practicality is limited.

### 2.5.6.2 Business Process Based Approach

Dignum and Weigand define an approach to organisation-based multiagent system design that is focused on business process based systems [81]. They
prescribe an organisation-based agent approach to assist with developing software for such business processes.

The focus of this approach is on taking an organisational perspective, with a focus on social behaviour. This is particularly applicable for agent systems that are intended to support existing organisations, or those that mimic a specific organisation. The composition of the organisation impacts on the relationships and activities of individual agents.

In this approach, there are four main properties that define agent societies: the purpose of the society, the structure of the society, interaction rules and communication languages, and norms.

A number of “coordination models” can be used to structure the agents’ interactions within the organisation. The coordination model used determines the openness, values, and facilitation roles of the agents in the organisation.

Low-level design is not considered, since it is assumed that the actual agents are “built somewhere else”, so implementation considerations are not described in any detail.

2.5.6.3 Other Organisational Approaches

A number of other organisational approaches have been proposed.

AALAADIN [82] defines the organisational structure at design-time in terms of roles. Agents must meet defined criteria before they can be admitted to an organisation (group). These criteria vary according to the kind of group being considered. Interaction between agents within a group is limited, with the valid interactions being limited by the roles of the two communicating agents.

The ROADMAP approach [83] is, like “extended Gaia”, derived from the original (non-organisation based) Gaia. It uses a tree to define the structure of organisations. Organisations may be dynamic, with “refactoring” occurring
when the agents within an organisation change. The organisational system as a whole is represented as a tree structure, with the root of the tree representing the entire multiagent system, other internal nodes representing organisations and sub-organisations, and the leaves representing agent roles.

### 2.5.7 Summary of design approaches

This section has provided a range of design approaches, each with their own advantages and disadvantages. These are summarised in Table 2.1.

<table>
<thead>
<tr>
<th></th>
<th>Detailed BDI Concepts (Plans, Goals, Beliefs)</th>
<th>Organisational Design</th>
<th>Life-Cycle Coverage</th>
<th>Low-Level Design Specificity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaia</td>
<td>Not detailed</td>
<td>Not specifically detailed</td>
<td>Analysis and high-level design only</td>
<td>No detailed agent design phase</td>
</tr>
<tr>
<td>Tropos</td>
<td>Basic coverage (Using Agent UML)</td>
<td>High-level architectural design</td>
<td>Requirements through to design</td>
<td>Limited modelling of reasoning</td>
</tr>
<tr>
<td>Prometheus</td>
<td>Covers all BDI concepts</td>
<td>Not explicitly, but defines communication protocols</td>
<td>No specific requirements elicitation process</td>
<td>Detailed. However, has a close modelling to JACK</td>
</tr>
<tr>
<td>Agent OPEN</td>
<td>Metamodel. Depends on “fragments” chosen.</td>
<td>Depends on “fragments” chosen.</td>
<td>Can be used to cover all life-cycle elements</td>
<td>Depends on “fragments” chosen.</td>
</tr>
<tr>
<td>Extended Gaia</td>
<td>Not detailed</td>
<td>Covered in detail</td>
<td>Analysis and high-level design only</td>
<td>No detailed agent design phase</td>
</tr>
</tbody>
</table>

Table 2.1. A summary of design approaches for multi-agent systems

Each of the processes has its own limitations, whether that be a lack of specificity in design, a strong coupling with a particular implementation platform, or a lack of lifecycle coverage. NUMAP addresses these issues by adopting a whole-of-lifecycle approach, with a focus on detailed modelling of both BDI concepts and organisational structures.
2.6 Applicability of multiagent systems

Multiagent systems are a unique paradigm for developing software, with a number of distinct advantages, as noted at the start of this chapter. The BDI model provides an especially promising technique for conceptualising software in terms readily understandable by humans. However, the possibilities for multiagent systems also lie in their synergies with other fields.

2.6.1 Web Services and Service Oriented Architectures

Web services have become a vital part of systems integration on the internet, with SOAP and WSDL [84] quickly becoming standards for allowing applications to discover, use and integrate functionality, using a service-based approach.

Service Oriented Architectures (SOA) are often described in many of the same terms as agents. Erl [85] defined the key principles of SOA to be: standardized service contracts (i.e. communication agreements); service loose coupling; service abstraction; service reusability; service autonomy; service statelessness; service discoverability; and service composability.

These principles share a significant degree of overlap with multiagent systems, particularly in relation to standardised service contracts, loose coupling, autonomy and reusability. Additionally, many of the infrastructure-centric capabilities of FIPA multiagent communication specifications [86] overlap with web service standards [87].

This could easily lead to a misconception that multiagent systems and web services are conceptually identical, but this is not the case [87]. Web services are generally stateless, they do not proactively pursue goals, and they do not possess the proactiveness and adaptiveness inherent to BDI agents. While agents and web services do possess some similarities, agents are unique in possessing intent and goal-directed behaviour.
Despite these differences, agents can interact synergistically with web services, using bi-directional integration to allow web services to be used by agents and vice versa [87]. This integration has been demonstrated via the creation of an automatic, bi-directional service integration gateway for seamlessly connecting web services and agents [88]. This provides a means for multiagent systems to take advantage of the current trend of increased web service availability.

While there is a real distinction between web services and agents, not least the added conceptual abstraction offered by BDI agents, integration with web services may assist with providing infrastructure and services for use by agents. Conversely, the proactive nature of agents may also provide a useful mechanism for dynamic web service composition.

### 2.6.2 Agents in the Cloud

The applicability of agents to service-based computing does not only extend to web services. A recent trend in computing has been the increasing popularity of cloud computing and “Software as a Service” (SaaS) [89].

An increasing number of companies are migrating their application software, middleware and information systems to cloud deployments. In many cases, this is done using a range of individual SaaS applications, which may not be designed with integration in mind [90]. A company moving their business systems into the cloud may end up using a variety of different SaaS products, for example, using separate systems for Customer Relationship Management; Document Management; email; and Supply Chain Management. Integration of such systems is, at present, very difficult, despite the pressing business need to do so.

Most cloud platforms provide a means for integrating services, called a Service Bus. However, these are not necessarily compatible between cloud vendors, and the actual task of integration can be difficult, requiring specific knowledge of application programming interfaces for each product. As a result, middleware,
called “integration Protocol as a Service” (iPaaS) is becoming popular for allowing integration of cloud services, and integration with on-premise deployments [90].

There is presently no standard approach for iPaaS integration, with a wide array of products available on the market to perform the basic task of service composition and integration. Multiagent systems provide a promising approach for this task, with their well-defined communications interfaces, and inherently service-oriented, collaborative nature.

Web services and cloud computing are technologies well suited for integration multiagent systems, and show great promise for future synergies with agent-based development. As such, any modern design process for multiagent systems should, at a minimum, be designed in such a way that external services such as those in the cloud or presented via WSDL can be readily integrated into multiagent systems.

### 2.7 Examples of multiagent systems

Multiagent systems have been used in real-world applications, in a number of fields. In particular, they have proven to be applicable to manufacturing, scheduling, business processes and workflow management, and planning applications. This section provides an overview of how agents have been used in these fields.

#### 2.7.1 Design and Manufacturing

Software agents have been applied to the field of design and manufacturing, for such purposes as industrial design, simulation, modelling, scheduling and control. Software agents are particularly well suited to this field, as industrial applications are often “modular, decentralized, changeable, ill-structured, and
complex”, all properties that are well suited to be solved by multi-agent systems [91].

During the industrial design process, multi-agent systems have been used for balancing design constraints. One example of this was a project called RAPPID (Responsible Agents for Product-Process Integrated Design) [92]. This project allows design constraints, such as weight and power use, to be “traded off” between design teams working on a product. It uses an agent-based marketplace to set prices on these constraints, with agents that represent each component buying and selling units based on these constraints. Using this system, the various tradeoffs in the design process are addressed in a market-oriented framework. For example, the design team could trade increased weight constraints for reduced power constraints.

Maintenance of supply chains is also a field in which agents can be used to enhance the usual manufacturing processes. Given a series of constraints, or conditions, agents can model the entire supply chain, and optimise the elements within it. This was done with DASCh (Dynamical Analysis of Supply Chains) [93], a system in which agents are used to represent the various components of the supply chain, such as companies, shipping and inventory control, and to perform an analysis of the behavior of each of these entities, to ensure that the system stays within the given conditions.

In some cases, an entire manufacturing system can be modelled using multiagent systems. Such modelling can expose inefficiencies within the system. This was done in one of the earliest applications of multiagent systems to manufacturing, a system called YAMS (“Yet Another Manufacturing System”) [94]. YAMS used a contract net with a hierarchy of agents to represent a manufacturing system. Agents are used to represent the overall shop, the individual workshops, and the devices in the workshops. The agent-based model of the manufacturing process uncovered issues with the hierarchical system being used. In particular, inefficiencies were discovered in
communications, with messages needing to constantly traverse the entire hierarchical structure.

Multi-agent systems have also been used for production scheduling. For example, an agent-based scheduling system has been detailed for a hot steel rolling manufacturing plant [95]. For this system, agents were designed to monitor the hot strip mill and the continuous caster, monitoring local information about these parts of the machinery, and responding to real-time events by adjusting the schedule in collaboration with other agents. The focus on using agents in such an application is their close coupling to the equipment being monitored – allowing the agents to react to events directly and rapidly. Agents have also been used in production planning to allow for production smoothing, planning and controlling production to minimise delays [96].

2.7.2 Monitoring and Control

As with manufacturing, agents can be used for monitoring and controlling complex systems. The close coupling of agents to physical entities within these systems makes them a natural fit for monitoring and controlling their operation.

Agents have been successfully applied in the area of electricity distribution management. A system using the ARCHON [97] framework has been used to aid control engineers in managing electricity distribution. This system allows remote operation of circuit breakers and reports on automatic switching operations as required by faults and alarms. The ability of agents to autonomously monitor alarm states and notify operators of events in real-time makes them well-suited for such an application.

In one of the more high-profile cases of multi-agent systems being used for monitoring hardware, a custom, rule-based agent system was used for monitoring the space shuttle [98]. The NASA Engineering Shuttle Telemetry
Agent (NESTA) monitored data from the Space Shuttle and allowed engineers to design rules so that they could be informed of matters of interest.

Agents have also been used to develop an architecture for an industrial transport system for controlling autonomous vehicles [99].

2.7.3 Business Applications

In business applications, agents may be used for areas such as modelling business processes, managing workflows, and project management. The ability of autonomous agents to be mapped to business units or individuals, allows for a natural relationship between software components and the entities to be managed.

In particular, workflow management involves managing discrete entities (roles or individuals), and coordinating them to perform a common goal. The JBees agent system does this, using agents for fluid business processes such as e-commerce [100]. In the JBees project, a workflow management system was implemented using multi-agent systems, and provides a mechanism for implementing decentralised and flexible business processes.

JBees uses Coloured Petri Nets to model business processes. Agents are structured into societies with social norms, and workflows are structured according to those norms. Resources (e.g. individuals) are divided into distinct roles, and each has various attributes and capabilities. Resource brokers negotiate with resources to allocate them to a task. By doing this, the assignment of individuals and resources within the workflow can be automated.

The concept of using agents to implement decentralised workflow systems was also used in the "ADEPT" [101] project at telephony provider BT. This project developed a business process management system for quoting customers on networks for telecommunications services. In ADEPT, agents representing business units communicate and negotiate to provide the services required for
quoting prices for services. The agents have heterogeneous information models and are designed around a service-oriented architecture. The designers argue that the use of agents allowed them to "develop a robust and flexible system that could not have been built using extant workflow technology", and enabled “different parts of the organisation (and indeed different organisations) to retain their autonomy of information and control", while providing loose coupling between inter-organisational business activities. [102]

Project management has also benefited from multiagent systems, particularly in geographically distributed projects. A multiagent system has been developed to support project management in such distributed environments [103]. In this system, activities and resources are represented by their own agent types, representing the activities to be done and the resources available within the project. A distributed multiagent system is used for scheduling and resolving resource conflicts, with negotiation between agents on geographically distributed computer systems taking place via message exchange. Within this system, no single computer contains all project-relevant data. The system consists of two parts: a small core program for running remote service agents and a collection of dynamic agents executing on the individual computer systems.

An agent-based conference planning application has been created using the “Scalable, fault tolerant Agent Grooming Environment” (SAGE) [104]. This system uses geographically distributed agents, along with web services and grid computing to support the process of conference planning. In this system, each agent represents a user, assisting them to ensure conference-related tasks are performed on schedule, and making recommendations based on their areas of interest.

In the financial arena, agent systems have been used for financial data mining in the F–Trade system [68]. F-Trade uses an open agent architecture to assist with the development and evaluation of financial trading rules. Agents were developed using an approach called Organization and Service-Oriented system
Analysis and Design (OSOAD). This allowed an open agent architecture to be created that supports the optimisation and testing of trading rules.

2.7.4 Decision Support and Knowledge Management

Agents have been used in decision support systems and knowledge management. This is particularly useful in situations where the information to be accessed is distributed, or information is incomplete. Agents, with their own internal belief states, can be used to collate the information from various sources and maintain a consistent model of information about a certain domain.

An agent-based decision support system has been developed for assisting with pest management, specifically for apple plant disease [105]. Agents were used due to the system dealing with distributed information. A systemic approach was used in development, where "a problem is analyzed in terms of all the knowledge, the data and the responsibilities it depends on".

Agent-based health care management has been implemented [97] that provides for coordinated health care management between various health care practitioners. An agent-based approach was chosen for this task due to the distributed nature of information, resources, problem-solving capabilities and responsibilities within the health-care domain. In this system, the agent-based approach also assists with the need to make decisions based on incomplete information and the need to pursue goals while monitoring changing circumstances.

A data warehouse using agent technology has been created for use in a Banking Business Intelligence System [106]. This system was designed using the Tropos methodology, and reported success, in a large part due to the formalisation of the system's early requirements.

Agents have been used within a type of Knowledge Management system called a "collaboration management system" [107]. This system, called the "Knowledge
Market", uses an open agent system to allow knowledge to be exchanged between users, while rewarding users for sharing knowledge.

### 2.7.5 Summary of Examples

Multi-agent systems have been applied to a range of applications, from manufacturing, to control, to business processes and knowledge management. In each of these applications, agents were best used when they could readily be mapped to real-world constructs, whether that be monitoring a particular piece of machinery, or assisting an individual within a business. In creating a software methodology for designing multiagent systems, this close mapping between real-world entities and the software agents should be a priority.

### 2.8 Goals for a Multiagent Design Process

Earlier in this chapter, the need for a process for developing multiagent systems was established. A number of existing processes were outlined, each with their own particular shortfalls, whether it be a lack of grounding in the real-world concerns faced by software developers, a tight coupling with a particular implementation environment, or an almost exclusive focus on a particular part of the software development lifecycle.

In order to address these shortfalls, a new process is required. However, as noted earlier in the chapter, designing a process is much like designing software – first, a set of requirements need to be established. These requirements, or goals, are defined in this section.

The aim of developing a new process for multiagent systems was to create a process that addresses the shortcomings of existing design approaches. A modern design process needs to be grounded in the real-world practicalities of developing multiagent systems, while remaining as “general purpose” as possible in terms of possible integration with external systems. The goal was to create a process that is detailed and prescriptive, allowing the software
developer to follow it without having to make leaps of logic to “fill in the blanks”, at stages where the process lacks detail.

Additionally, it was decided to attempt to include mechanisms within the process to increase flexibility, so it may be used with agent design techniques outside of its primary target of BDI multiagent systems. The aim was to ensure the process is constructed in such a way that it can be easily tailored to support alternative agent design approaches, without losing any of the advantages that come with being specifically designed for BDI systems. It was decided that a modular approach would be required to achieve this.

Finally, the process should be centred on the notions of “human action” from the earliest stages of development. BDI multiagent systems offer significant advantages around the level of abstraction they offer. Goals, agents and plans are natural concepts to use for thinking about software design.

These concepts should be used from the earliest stages of development, right through to detailed design and implementation. Rather than introducing new concepts at each stage of development, the design process should revolve around refinement of these key concepts as the developer proceeds through the phases of the process.

These aims were formalised into the following six goals, which shaped the design decisions during development of the process:

1. End-to-end support for the software development lifecycle.
2. Grounding in real-world agent design concepts.
3. Goal-based requirements engineering.
4. Modularity and process tailoring.
5. Practicality and detail.
6. Inclusion of a comprehensive support tool.
The remainder of this section describes these goals in detail.

2.8.1 End-To-End Development Support

One of the features found lacking in existing processes was full support for end-to-end software development. Some processes aim to provide a flexible approach to developing software by keeping the design phase of development as broad as possible.

For example, the Gaia process⁴ focuses on a high-level approach to agent design, leaving the detailed design process to be considered during implementation. The aim of this approach is to ensure the developer is “free to realise the services in any implementation framework” [20]. While this gives the developer freedom to choose the method of implementation, it also increases the complexity of implementation, and reduces its consistency when compared to a more prescriptive design phase.

The Tropos [21] framework provides for a very detailed requirements elicitation phase, but lacks the same detail when it comes to modelling complex internal reasoning behaviour of agents. As mentioned earlier in this Chapter, Tropos has been described as being “not suitable for sophisticated software agents requiring advanced reasoning mechanisms for plans, goals and negotiations” [73].

In both of these cases, the agent design processes do not provide detail for the entire lifecycle of agent development. This gives them the advantage of generality - they avoid limiting the process to a single agent development technique. However, this approach makes the task of implementing the system

---

⁴ When referring to “Gaia”, we refer to the original process. There have been several modifications to Gaia proposed by various authors. However, the original process is still the most influential, and is most commonly referenced in the literature. Therefore, it will be used as the basis for comparison.
extremely difficult, as the design is not fully specified at a lower-level. Many design decisions are not spelled out at all during the development process, leaving the implementation open to interpretation.

In developing a new multiagent process, a core goal was to ensure that the process is as prescriptive as possible, allowing developers who are new to multiagent system design to follow the process, and produce a system that is well-structured. This requires an end-to-end approach to be taken. The process should guide developers through all steps of development, from early requirements analysis through to implementation.

A goal for a new multiagent design process is therefore to provide software engineering practitioners with guidance for all stages of development.

2.8.2 Close Mapping to Real-World Implementation

The ultimate goal for any software design process is to guide users through the process of creating well-designed software, to be implemented in a suitable language and environment.

The aim of the NUMAP design process is to create systems comprised of software agents. In particular, its focus is on using the BDI paradigm for such agents. A number of implementation environments exist for creating and deploying BDI agents, including popular options such as JACK [65] and Jadex [39] (described in Chapter 3).

These BDI implementation environments follow the basic structure established by PRS [67], with agents defined in terms of the plans, beliefs and goals. A design process for creating BDI multiagent systems needs to be based on these concepts, if it is to be used successfully for creating multiagent systems within them.

Effective multiagent system design relies on properly specifying both the high-level architecture and organisation of the system, along with the internal
reasoning of the individual agents. Attempting to define agents without taking into account the concepts used for defining both these aspects will invariably lead to an incomplete specification of the system, making implementation a difficult task.

Therefore, the lower-level design aspects of the process must match real-world agent implementation environments as closely as possible. The implementation stage of the process should be a straightforward one, with little ambiguity in the design. Implementation should be a relatively “mechanical” process, with the design specified in enough detail that teams of developers can successfully cooperate in their development effort.

With this in mind, a goal of a new multiagent design process is to provide a close mapping to real-world agent design concepts.

2.8.3 Goal-Based Requirements Engineering

Accurately identifying the requirements for a software system is an essential activity, albeit one that is complex from a conceptual and practical standpoint. In attempting to identify requirements using conventional elicitation techniques, system goals are often seen as an important precursor to the actual requirements – they are used as a means of surfacing the requirements of the system [108].

A newer approach to requirements engineering involves making goals a formal part of the process, whether being used as an aid for identifying requirements, or as a central part of the requirements engineering method [109]. Techniques using this approach include the KAOS goal-oriented requirements framework [110] and Anton’s Goal Based Requirements Analysis Method (GBRAM) [111].

Goal-based requirements engineering techniques are a good fit for agent design, with their focus on stakeholders (or agents) and goals as primary concepts for specifying requirements [112].
The use of such techniques during the requirements engineering phase of the design process is therefore desirable, so an aim of a new multiagent design process is to support the use of goal-based requirements engineering techniques.

2.8.4 Modularity and Process Tailoring

While the focus of this work is on creating a design process for systems designed with the BDI approach, and using a goal-based requirements engineering technique, these are not the only alternatives for developing multiagent systems.

Non-BDI approaches to agent design do exist, such as the subsumption [56], and swarm [60] approaches detailed earlier in the chapter. Also, one of the most promising avenues for the use of multiagent systems is in integration with web services and cloud applications. The ability to seamlessly model these services within the system as if they were themselves “agents” is desirable.

Additionally, a variety of requirements elicitation techniques exist, and whilst goal-based approaches seem the most appropriate for designing multiagent systems, some software engineers may prefer more conventional approaches.

It would be beneficial to allow the process to be tailored. Different projects may use differing agent technologies, and different project teams will have experience with different requirements elicitation techniques. Ideally, it should be possible to tailor the process to support any requirements elicitation technique, and support various agent design approaches.

If the process were to be monolithic, this would be a difficult task. Without adopting the approach taken with other processes, and creating a very generic, non-specific detailed design phase, it would be difficult to avoid constraining developers to using particular techniques.

One alternative would be to use an approach such as the OPEN Process Framework [113]. This is a method engineering approach that constructs
processes using reusable fragments [114]. By creating a library of fragments, and constructing a process from them, a high degree of customisation and tailoring would be possible. This has been applied to agents in Agent OPEN[115].

However, as the focus of this work is to create a coherent, well-structured process that is usable by any software engineer, regardless of their degree of familiarity with agent-oriented systems or method engineering, it was felt that such an approach was not appropriate at this stage.

A simpler, more straightforward approach to process tailoring is preferred. The process will therefore be specified in a modular fashion, with each phase being a cohesive component that can be replaced or extended with minimal impact on the rest of the process. This is particularly important in regards to the phases for requirements engineering and detailed agent design.

A core, “default”, process is required, that will support the Goal Based Requirements Engineering approaches detailed above, along with a detailed design phase catering for BDI-based agent implementation environments.

Software engineers will then able to tailor the process to support other requirements elicitation techniques, agent design techniques, and implementation environments.

This approach strikes a balance between the straightforwardness of monolithic processes and the flexibility of meta-processes such as OPEN.

A goal of the process is therefore to support modularity and process tailoring.

2.8.5 Practicality and Detail

For the process to be usable by a wide audience, it needs to be designed in a manner that is easy to follow, providing sufficient detail at each stage to allow the system to be specified fully.
The process must be practical and detailed enough to be implemented without ambiguity. All stages of the process should provide sufficient detail to ensure that activities are specific and repeatable.

The concepts used in the system must be well-defined, with clarity about what information needs to be specified at what stage, and in what format that information should be specified. This is best achieved using a forms-based approach, by which the user specifies the required information at each stage in a specific manner, as defined by the process.

A goal of the process is therefore to ensure the process is practical and detailed at each step.

2.8.6 Comprehensive Support Tool

In order to aid the developer in following the process, a comprehensive support tool would be highly advantageous. This should provide functionality to assist the developers in specifying the required information at each stage of the process.

It is desirable for the process to also allow for automatic creation of documentation and for code templates to be generated from the design.

It is therefore a goal to provide a support tool that aids the user in following the process.

A modern design process for multiagent systems should be designed in such a way as it satisfies these six goals.

2.9 Conclusion

This chapter provided background and context for multiagent systems, along with an overview of the variety of agent-based process available and their
characteristics. A number of real-world multiagent systems were also presented.

Additionally, the goals for a modern multiagent design process were established.

In the next chapter, the advantages of the BDI model are discussed, and an overview of software tools and implementation environments that support BDI multiagent systems is provided.
This chapter outlines the advantages of BDI multiagent systems, and provides an overview of the tools and implementation environments that are available to develop such systems.

3.1 Features of BDI Systems

Before discussing BDI multiagent systems, it is worthwhile to list some of the attributes they generally possess. The following features are common among most implementations of BDI agents:

- **Autonomy, Temporal Continuity and Social Ability**: An agent is an autonomously executing software component, performing a specific task by collaborating with other agents.
- **Parallelisation**: Each agent executes in parallel, with an independent thread of execution.
- **Encapsulation**: Agents are seen by other agents in their environment solely as “service providers”. The internal logic of how they deliver those services is hidden, and may involve as simple or as complex planning as is required to do so.
- **Pro-Activeness**: BDI agents have a planning component to their pro-activeness (either via pure planning or a hybrid approach). Agents are defined with a library of predefined plans, which specify the actions they may take.
- **Goal-oriented**: Agents are tasked with achieving or maintaining given goals, using their predefined plans.
- **Adaptiveness**: BDI agents may adapt in order to achieve their goals. Their internal logic can change over time. The exact nature and extent of this
adaptiveness depends on the individual agent. However, BDI agents generally have the ability to recover from plan failure:

- **Safe fail**: Agents may change their planning strategies based upon failure to achieve a goal.
- **Communicative Collaboration**: Each agent communicates and collaborates with other agents.
- **Role**: An agent plays a specific role within the system and provides services to other agents based upon that role.
- **Environment**: A BDI agent is embedded within an environment, either physical or virtual.
- **Reaction**: An agent can act based on external stimuli, either immediately by a reactive approach, or through updating its beliefs and adjusting goals accordingly.
- **Sensing**: An agent is embedded within its environment, and includes some form of “sensor” for detecting changes to its environment’s state. This allows it to change its plan selection in response to changes to its environment.
- **Beliefs**: An agent updates its beliefs based upon the sensed state of the environment. These beliefs provide a “fuzzy” depiction of the state of the environment, and need to be updated to be kept current.

### 3.2 Advantages of BDI multiagent systems

The preceding chapter presented the background of BDI multi-agent systems, along with techniques for developing such systems, and examples of their use were presented. In this section, the advantages of such a technique are listed.

As a unique approach to designing software, BDI autonomous agents provide a number of specific advantages and opportunities for the developer. One of the most important is the fact that such agents provide an improved abstraction for design. Beliefs, desires and intentions are a natural conceptual metaphor for the operation of software, based on notions of how people reason about action [11].
Software components may be thought of as being individual autonomous entities, with their own motivations and knowledge. This higher degree of abstraction may assist developers when thinking about complex systems, which has been referred to as the ability for agent-oriented design to operate at the “knowledge level”, rather than the “symbolic level used by other paradigms [36].

Additionally, BDI agents provide a large degree of flexibility. Developers can change the dynamics of a system by reconfiguration of the plan library and agent population [116] rather than requiring extensive rewriting of code. This is especially advantageous for software dealing with business processes and manufacturing, which needs to be flexible [91]. A change in the way plan selection is configured, or the addition of new plans to satisfy an existing goal can improve system functionality with minimal modification being needed of existing code.

In systems such as business process management, where there is a need to monitor execution of tasks and services, and to ensure conditions are met, BDI agents may be a very suitable approach [101]. Rather than the software being required to continuously check if some condition has arisen, agents may simply monitor the system for that condition, and perform tasks in response.

Agent-oriented design provides an ideal means for designing concurrent systems. Each agent represents a separate thread of execution, providing an inherent facility for concurrency within the system, a very suitable approach for modern computer hardware, with its increased capacity for parallelisation. This autonomous nature also improves encapsulation within agent systems. Each agent is an independent component, and can be considered a “black box” [117] from the perspective of other agents within the system.

Complex environments are also well suited to an agent-oriented approach [2]. The adaptive nature of agents means that complexities within the environment can be handled at run-time. The flexibility through which agents can achieve
their goals allows for unexpected occurrences in the environment to be better dealt with.

Agent-based systems also provide a means of distributed control over a system. Rather than having one central thread controlling the flow of execution, agent-based systems allow for control to be distributed among all the entities in the system. Co-operative agents can engage in team-learning [118] and distributed problem-solving [119] to engage in a collaborative solution to a problem.

Legacy systems may be easily integrated into multi-agent systems, with an interface agent acting as a “wrapper”, allowing the system to be fully encapsulated as an agent [40]. The agent’s services will be defined to correspond to the functionality of the legacy system, allowing it to be transparently integrated within an agent community.

The use of agents can also enhance platform independence [36]. Most agent runtime environments are cross-platform, and allow agents executing on different platforms to communicate. An agent system may be composed of several discrete nodes, each executing on a different platform.

Agents are well suited to agile methodologies, such as “extreme programming”, or where there are unclear or change-prone requirements [120]. Additional requirements can be addressed by combining services from existing agents, and new agents can be added to the system with minimal or no changes being needed to existing agents. Due to the small size and complexity of individual agents, refactoring is made easy.

In situations where graceful recovery from failure and response to real-time perturbations is required, agents may offer a suitable approach [41]. If an agent fails within the system, other agents using its services may seek out other service providers within the system to allow recovery. This allows graceful performance degradation as nodes fail [121]. If a node fails, agents that were
using the services provided on that node can attempt to find other agents offering the same services.

It has also been argued that agent-oriented design allows for enhanced maintenance and reusability [36]. Agents can be reused when the services they provide are again required, and the system can be maintained by amending individual agents, while leaving the rest of the system untouched.

Product-line software may also benefit from the use of agents [122]. Shared features of the software product line can be provided by common agents, while specific functionality may be added via other agents. This provides a mechanism for rapid development of software in keeping with the software product line approach.

Agent systems allow distributed applications to work with limited information at each node [119]. The ability to decompose systems with discrete agents allows those agents to execute on different physical nodes. This is especially useful when different nodes have access to different information, which may be impossible to share among nodes due to resource constraints or business rules.

These advantages offered by software agents can only be exploited with the proper processes and tools for creating robust multiagent systems. The remainder of this chapter outlines these tools.

### 3.3 Tools for developing multiagent systems

In order to effectively develop multiagent systems, tools are required to support the development process, and to provide the infrastructure required for the execution of multiagent systems.

BDI Agent implementation environments need to provide support for specifying plans, defining beliefs, passing messages, registering agents for communication, event handling, and service definition.
A number of such implementation environments, or frameworks, are described in this section.

### 3.3.1 The Procedural Reasoning System

The first practical software system to be based on BDI principles was the Procedural Reasoning System (PRS) [67], developed by Georgeff and Ingrand.

This system has been influential. Many modern BDI implementations, such as Jadex [39] and JACK [65], are based on the principles identified in PRS.

One of the primary aims of PRS is reasoning in a changing environment. It aims to allow easier development and maintenance of software systems that are embedded in a complex, changing environment [11]. This is accomplished by supporting partial planning and reactive action, while providing reflective capabilities that allow for higher-level reasoning.

The key aspects of the PRS system are: a database containing *current beliefs* about the world; a set of goals that the agent attempts to realise; a set of *Knowledge Areas*, or KAs (plans), describing the actions required to achieve particular goals; and a list of plans that have been selected for execution stored in an *intention structure*. The execution of KAs is controlled by an *interpreter* that selects KAs based upon the system’s beliefs and goals, and places these on the intention structure to be executed.

#### 3.3.1.1 Beliefs

The PRS database of beliefs represents knowledge in first-order predicate calculus. This includes dynamic observations about the current environment and conclusions derived from these observations. Static facts about properties of the application domain are also stored.
3.3.1.2 Goals

Goals in PRS are “expressed as conditions over some interval of time” using state descriptions [67]. The system provides a number of types of goals, including: achievement; maintenance; and testing for a condition. A goal is achieved if the system state is changed in such a way that it satisfies the goal description.

3.3.1.3 Knowledge Areas (Plans)

The KAs in PRS consist of both a body and an invocation condition. The body is essentially a plan, which consists of actions to be done and sub-goals to be achieved. The plan is executed by performing these actions and achieving the sub-goals from start to end. The invocation condition indicates to which conditions the KA is relevant. It describes the events that must occur for the KA to be executed, these can be either goal-directed, responding to a posted goal, or reactive, responding to some change in beliefs.

3.3.1.4 Intentions

PRS intentions are similar to processes, and contain an initial KA as well as all the child KAs that are used by it. The intention structure consists of all intentions that the system is either executing, or has chosen for execution in the future. Intentions may be active, suspended or awaiting an activation condition. PRS commits to intentions even if circumstances change. If a goal fails, the goal will be re-established and another attempt made. This is repeated until all applicable KAs have been tried.

3.3.1.5 Interpreter

The system as a whole is managed by the PRS interpreter. This is responsible for choosing a KA to execute from the set of KAs that are applicable in terms of the system's goals and beliefs. If a KA is chosen to be executed as the result of a top-level goal, it is placed on the intention structure as a new intention.
Otherwise, if it is selected to fulfil a sub-goal, it is added to an existing intention structure.

Deduction is not performed by the interpreter. Rather than use a “compatibility filter” and “filter override mechanism”, as previously described in Bratman’s approach, the decision making processes used by PRS are hard-coded into the interpreter. Any deliberation is performed by the _metalevel_ functionality of PRS, which provides a mechanism for selecting KAs, altering intentions and reacting to goal failure.

### 3.3.1.6 Usage and importance

PRS has been successfully used in several real-world applications. Its success in these applications has been due to: its partial planning strategy, its reactivity, the use of procedural knowledge in KAs, and its metalevel functionality. [67],

The PRS system provides a concrete platform for the implementation of BDI multiagent systems. The basic PRS-style architecture can be seen in more recent BDI systems such as Jadex [39], although recent systems use the more familiar terminology of “plans” rather than “knowledge areas”.

PRS demonstrates a number of important capabilities of agent systems, including the need for both reactive and proactive functionality in changing environments, and the ability of plans to be individually designed by domain experts, and integrated into a whole system. The applicability of iterative design to agent-based systems is also demonstrated by PRS, with it providing the ability for agents to be refined by introducing plans for increasingly specific cases over time. This approach to iterative development has similarities to agile development, and in particular the notion of driving product development through prototyping.

One of the main instances of a PRS system is the OASIS air traffic management system [123]. The system was non-deterministic, embedded in a complex,
unpredictable environment that can only be sensed locally, with reasoning dependent upon the environment.

3.3.2 Modern BDI Systems

The PRS had a large influence on future BDI multiagent systems. The concepts of goals, plans (or knowledge areas) and beliefs became the central building-blocks of future BDI systems and implementation environments.

The two most notable examples of modern BDI platforms are JACK and Jadex.

3.3.2.1 JACK

JACK [65] is an agent development environment and implementation platform for creating BDI multiagent systems. It is built upon Java [124], providing language extensions specifically for designing multiagent systems.

JACK is designed to be an agent-oriented extension to Java, much as C++ is an object-oriented extension to the C language [125]. Therefore, JACK supports all features of Java, with agent-focused language extensions. JACK provides its own compiler, which converts JACK code into pure Java, enabling it to be run on any Java Virtual Machine.

JACK agents are defined using six different kinds of source file, corresponding to the core base classes within a JACK platform: agents, capability, beliefset, event, plan, and view.

Each of these source files is defined as an extension to the Java language, including the full set of Java syntax, but extended with additional directives, preceded by a “#” sign. Additionally, several new keywords are added to the JACK syntax over Java. JACK agents rely upon a Java library to implement their functionality, and are translated to Java bytecode by the JACK compiler.
**Agent**

Agents are the main components of the JACK system. These are software components that exhibit both reactive and proactive behaviour. Each JACK agent is defined in terms of its beliefs (its data-set), a set of events to respond to (reactive behaviour), a set of goals it may attempt to fulfil (pro-active behaviour), and a set of plans specifying how to go about fulfilling its goals or events.

JACK agents are defined with the following format:

```java
agent RetailerAgent extends Agent {
    #private data OrderStatus orders();
    ...
    #handles external event CustomerRequest;
    ...
    #sends event ShipmentInfo shipmentinfo;
    ...
    #uses plan ShipOrder;
    ...
    RetailerAgent (String name)
    {
        super(name);
        ...
    }
}
```

**Capability**

Capabilities group together a set of events, plans and beliefsets for use by an agent. They are a means of structuring reasoning components to allow reuse, while encouraging modularity and encapsulation.

Capabilities may be “plugged in” to an agent to provide a set of functionality. The development of “capability libraries” consisting of various capabilities each offering a predefined set of functionality is encouraged.

Capabilities are defined in a manner almost identical to agents, albeit using the “capability” keyword.
BeliefSet

BeliefSets define the data set of the agent, i.e. the internal beliefs of the agent about its world. All beliefs an agent has for a particular beliefset relation are represented as a tuple. Tuples may be true or false, representing whether the agent believes that the statement is a true one or a false one.

JACK supports two different forms of beliefset: “Closed World” and “Open World”. Closed World beliefs assume that the agent is operating in an environment where all beliefs can be considered to be either true or false. The agent stores data for those beliefs it believes to be true, and all other beliefs are assumed to be false. Open World beliefs may encompass an unknown state. Therefore, the agent stores both true and false relations. Any beliefs not stored are assumed to be unknown.

JACK beliefsets are defined with the following format:

```
beliefset OrderAmount extends ClosedWorld {
  #key field intorderid;
  #value field intorderqty;
  #indexed query get(intorderid, logicalintorderqty);
}
```

Event

JACK events define the situations in which an agent will take action. When an event occurs, the agent performs a task to handle it. In the absence of events, a JACK agent will sit idle.

There are two main forms of events in JACK – normal events and BDI events.

- A normal event represents reactive behaviour. When such an event is received, the agent selects a plan that is relevant (i.e. that can handle the event) and applicable (i.e. its context evaluation succeeds), and executes the plan. If the plan does not succeed, the event is considered to have failed.
A BDI event represents proactive behaviour. When such an event is received, the agent commits to achieving it. The agent engages in meta-level reasoning to select an appropriate plan, and executes it. If the plan does not succeed, the agent can reconsider other applicable plans and use one of them instead.

JACK events are defined with the following format:

```java
event UpdateStockLevel extends Event
{
    Product product;
    int stocklevel;

    #posted as updateStockLevel(Product p, int sl)
    {
        product = p;
        stocklevel = sl;
    }
}
```

**Plan**

Plans are a sequence of actions, stored in the agent’s *plan library*, which it can use to achieve its goals or handle events. Each plan is mapped to handling a single event, defined by its *handles* declaration.

A plan may specify a *relevant()* method, to assess for which particular instances of events it is relevant. Additionally, it can specify a *context()* method to evaluate whether it is applicable to a particular occurrence of the event.
JACK plans are defined with the following format:

```java
plan ProcessOrder extends Plan {
  #handles event OrderReceived orderReceived;
  // Define events, data, context and relevance here
  body()
  {
    ...
  }
}
```

**View**

Views in JACK provide the capability to query a dataset or beliefset. They allow for complex queries across beliefsets, or even to external systems.

In summary, JACK provides a mature, stable environment for implementing multiagent systems. It has a full-featured syntax for specifying BDI systems, and has been used in numerous real-world projects [65].

### 3.3.2.2 Jadex

Jadex [39] provides a software framework for implementing BDI agents. It began as a BDI-oriented extension of the earlier JADE [69] agent middleware, but over time has been altered to become a self-contained agent implementation environment.

Jadex provides a runtime platform for agents, supporting a BDI reasoning engine, message transport between agents, and directory facilitation for registering agents and services. Jadex agents are defined in XML, with agent plans and beliefs being defined as standard Java objects.

Jadex agents are defined using an Agent Definition File, in standard XML. This defines the agent’s beliefs, goals, plans, capabilities and the events to which it will respond.
Beliefs are defined in terms of standard Java classes, either as a singular belief (representing a single instance of the relevant class), or as a beliefset (representing a collection of the relevant class).

Goals are specified as either maintenance goals, keeping a given parameter or parameters within an acceptable limit, or as achievement goals, aiming to bring about some state in the system. Plans are defined within the Agent Definition File by specifying the trigger condition for the plan to be invoked (either a goal or an event). Individual plans are implemented as Java classes, inheriting from a standard Plan class.

Events, including incoming message processing, are also defined in the Agent Definition File, allowing the agent to have reactive functionality. When an event is triggered, an agent will immediately invoke a plan if one is associated with that event.

Snippets from an example Jadex Agent definition file are shown in Figure 3.1.

Jadex provides a feature-complete platform for implementing multiagent systems, and is one of the few agent frameworks that have been in active development for some time. Jadex began development in 2003, and the project is still active as of 2013. As such, it is a promising target for agent development.
Figure 3.1 – Jadex Code Snippets

3.3.3 Other agent frameworks

A number of other frameworks for developing agents exist, although they have not seen widespread use in real-world projects to the extent of JACK or Jadex.
3.3.3.1  JAFIMA

Kendall et al define a framework, called JAFIMA [126], for developing intelligent and mobile agents, using a multi-layered architecture and object-oriented design patterns.

The JAFIMA framework is divided into seven layers. These are mobility, translation, collaboration, actions, reasoning, beliefs and sensory. The sensory layer updates the agent’s beliefs based on data sensed from the environment. Sensing is achieved through a sensor that interacts with interface classes specific to the domain. The beliefs layer handles primitive and composite beliefs (i.e. data structures). When beliefs are updated, the reasoning layer is contacted.

The reasoning layer is used to decide what actions an agent will take, by the use of expressions and plans.

The action layer executes the plans that the reasoning layer has decided upon. Instantiations of plans are referred to as intentions, and run in their own thread. Intentions may also be instantiated by collaborations or reactions. This layer is also responsible for scheduling and prioritising actions. The action layer primarily uses the capabilities of other layers.

The collaboration layer negotiates with other agents and is responsible for providing and using services. It provides different communication protocols, handles requests, and provides facilities for replying to messages.

A single mobility layer is common between multiple agents. It allows agents to migrate hosts transparently via cloning.

The authors of JAFIMA claim that "in comparison to other frameworks for agent based systems, JAFIMA is more comprehensive, addressing more aspects of agency" [126].

JAFIMA combines a process for agent development with a Java-based architecture. While the support for a whole range of agent functionality, from mobility to reasoning is impressive, the platform has not seen any real-world implementations, outside those described by its creators.

### 3.3.3.2 ZEUS

ZEUS [127] is an agent building toolkit, developed by BT Laboratories, that aims to provide a graphical environment for creating multiagent systems. ZEUS provides a *"generic, customisable, and scaleable industrial-strength collaborative agent building toolkit"* [128], using a visual environment for rapid development.

ZEUS classes are grouped into three functional groups:

- **Agent Component Library**: Planning and scheduling, communication, social interaction, agent concepts.
- **Agent Building Tool**: Visual agent creator, auto code generator, legacy system APIs.
- **Agent Visualisation Tool**: Society viewer, reports tool, statistics tool, agent viewer, control tool.

Associated with the ZEUS toolkit is a development methodology, specifically tailored for creating agents with the toolkit.

A number of systems have been created with the ZEUS toolkit, however the software is no longer in active development.

### 3.3.3.3 SoFAR

SoFAR, the *Southampton Framework for Agent Research* [129] is an agent framework that applies ideas from distributed computing to agent research. The SoFAR framework focuses on the use and provision of services by agents.
Proactivity within the system is provided by the opportunistic use of services by agents.

Communication within SoFAR is based upon the startpoint/endpoint paradigm. An agent announces what kind of messages it can handle via a registration service. Actual communication details are abstracted away from the implementer. SoFAR uses XML for defining agents and ontologies, with the aim of separating specification from implementation. The XML file is later converted to a Java template with common agent code to reduce programmer effort. Use of SoFAR has been restricted to the domain of Distributed Information Management, for the management of information within a distributed environment over its entire life-cycle.

### 3.3.4 Summary of Tools

The above systems are summarised in Table 3.1.

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Initial Development</th>
<th>Tool Type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedural Reasoning System</td>
<td>Late 1980s</td>
<td>BDI (Reasoning) Platform</td>
<td>First practical software system based on BDI principles. Proved influential for future systems such as JACK and Jadex.</td>
</tr>
<tr>
<td>JAFIMA</td>
<td>Mid-1990s</td>
<td>BDI Implementation Framework</td>
<td>Java-based framework. However, few real-world applications developed.</td>
</tr>
<tr>
<td>ZEUS</td>
<td>Late 1990s</td>
<td>BDI Implementation Toolkit</td>
<td>Java-based toolkit for multiagent systems. No longer in active development.</td>
</tr>
<tr>
<td>SoFAR</td>
<td>Circa 2000</td>
<td>BDI Implementation Framework</td>
<td>Framework for multiagent systems developed at University of Southampton, and primarily used in research there.</td>
</tr>
</tbody>
</table>

Table 3.1. A summary of tools for building multi-agent systems
Of the agent frameworks surveyed, JACK and Jadex are the most mature and robust. A goal of NUMAP was therefore to be well-suited for development with these implementation platforms.

3.4 FIPA Specifications

In designing and specifying multiagent systems, a standardised means is required for describing basic concepts such as how agents interact and the mechanisms for their communications.

The Foundation For Intelligent Physical Agents (FIPA) [130] has defined a series of specifications for these purposes, focusing on way by which autonomous agents can communicate and interact.

As of early 2013, twenty five specifications have been standardised by FIPA. The majority of these specifications fall into two categories: message transport and representation; and the definition of communicative acts and interaction protocols to be followed by agents performing particular types of communication.

The standardisation of interaction protocols for agents is vital, as well defined patterns of communication are essential for the organisation of individual agents into a coherent, well functioning system.

3.4.1 FIPA Communicative Act Specification

FIPA have created a library of “Communicative Acts”, defining a suite of basic interactions that agents may perform.

The specification of communicative acts is an essential part of formally structuring agents into well-functioning systems. The definition of such acts allows a standard set of interactions to be used by agents and provides for the reuse of common communication actions (referred to as “macro communicative acts” by FIPA). Communicative acts are the basic building blocks for agent
communication. They are defined using a formal model, detailed in the Communicative Acts Specification [131].

These communication protocols provide the basis for the basic communication primitives used for messages between agents in modern BDI systems such as Jadex [132].

The FIPA Communicative Act Standard defines a number of acts that may be performed between a sender agent and a receiver agent. The defined acts are, in alphabetical order: Accept Proposal; Agree; Cancel; Call for Proposal; Confirm; Disconfirm; Failure; Inform; Inform If; Inform Ref; Not Understood; Propagate; Propose; Proxy; Query If; Query Ref; Refuse; Reject Proposal; Request; Request When; Request Whenever; and Subscribe.

A number of these acts relate to proposals to perform a given action. The Call For Proposal act initiates a negotiation process, calling for proposals to be made to perform a specified action. The Propose act is a general purpose act for initiating a proposal or responding to a Call For Proposal. The Accept Proposal and Reject Proposal acts provide appropriate responses to a previously submitted proposal.

Another grouping of acts relates to providing information about a given proposition. The Inform act indicates that a given agent holds a belief that a certain proposition is true, and wishes for another agent to come to hold the same belief. The Confirm act indicates to the receiver that a proposition is believed to be true in the situation where the receiver is known to be uncertain about the proposition, and the Disconfirm act indicates that a proposition is believed to be false in the same situation. A Subscribe act requests that an agent be informed whenever the value of a given proposition changes.

Variants on these acts are Inform If, which indicates that an Inform act for a proposition, or its negation, is to be performed based on the belief about the given proposition, and Inform Ref, which informs the receiver about the object
that corresponds to a descriptor, such as a name. Agents may request information about a given proposition or object using the *Query If* and *Query Ref* acts.

One of the most common categories of communicative acts relates to requests for agents to perform an action. An agent may request a recipient to perform a given action using the *Request* act. Alternatively, they may request that an action be performed when a given proposition is true with the *Request When* action, or for it to be performed every time a proposition is true with the *Request Whenever* action. An agent may respond to a request for action with an *Agree* act, or with a *Refuse* act. Once a request has been made, the requesting party may use a *Cancel* act to indicate that the action is no longer required, or the party performing the action may use a *Failure* act to indicate that an attempt to perform the action failed.

A number of basic acts can also be used for communications. An agent may use a *Not Understood* act to indicate a given message could not be interpreted. Additionally, the *Propagate* and *Proxy* acts allow an embedded message to be relayed to another agent or group of agents.

These communicative acts form the basic building blocks for all FIPA-compliant agent communications, and are hence vital for constructing well-functioning multi-agent systems. Due to the standardisation of these acts and their ubiquity in modern multi-agent systems, they necessarily form the building blocks for defining communications in a multi-agent design process.

### 3.4.2 FIPA Interaction Protocol Specifications

While the above communicative acts provide the basic building blocks for agent communication, there are a number of more complex agent interactions that are performed regularly.
By defining these actions as FIPA standards, common situations such as requesting an action or subscribing to information updates can have well-defined interaction patterns, allowing better code reuse and interoperability between agents.

FIPA provides a number of Interaction Protocol Specifications for such purposes. Each of these specifications defines a protocol flow, consisting of a series of agent communicative acts, which allows a well-defined interaction to be undertaken between two agents.

A number of such interaction protocols have been specified for performing requests, queries, establishing contracts, performing auctions, undertaking message brokering, and subscribing to notifications.

For example, the Request Interaction Protocol[133] allows an agent to request another agent to perform some action. The agent that initiates the interaction sends a message to the other participating agent via a Request communicative act. This requests some action be performed. The recipient of this message may then refuse this request, or agree to perform the specified act. If it is the latter, a notification will be sent at the end of the action notifying either: failure; completion of the action; or a result for the action, depending on what was requested and the outcome of the action.

An example representation of this protocol, using an AUML [72] sequence diagram (based on that from the FIPA standard) is presented in Figure 3.2.

\[\text{\footnotesize AUML} \] is an agent-focused extension of UML that is proposed by FIPA for modelling agent-oriented concepts.
Such communicative acts facilitate complex interactions between agents, and allow reuse of code for common interactions.
3.5 Summary

The BDI architecture, and the modern BDI systems that are built upon it, provide a mature, coherent model for developing software in an agent-based framework.

As BDI agents are intended to adapt during execution, pursuing their goals by any number of plans, agents are especially suited to environments that change in complex or unpredictable ways. Changes in the environment can be handled at run-time, by altering the plans selected for pursuing specific goals.

BDI agents are designed using notions of behaviour that are instantly familiar to software designers, due to their basis on human reasoning. When we think of a system, we often naturally confer human notions onto it, thinking “what does this software need to do”, or “what information does this component want to know”. BDI Agent-oriented design leverages this tendency in its use of human reasoning as the basis of its design vocabulary.

The maturity and completeness of the BDI model has led to it being one of, if not the most common, means used in the reviewed literature for describing the internal logic and state of agents within a multiagent system. It provides a powerful, intuitive mechanism for this purpose. As such, it has been the primary focus of this work.
Chapter 4 The NUMAP Design Process

4.1 Overview

The need for a process to guide the development of multiagent systems was established in Chapter 2, and the goals for a successful process to satisfy this need were defined.

In this chapter, the Newcastle University Multi-Agent Process (NUMAP) is presented.

4.2 Goals for a Multiagent Design Process

The NUMAP multiagent software design process has been designed to satisfy a number of unique requirements. In meeting these requirements, it addresses shortfalls of existing processes and provides a modern approach for the development of multiagent systems.

The goals to be met, which shaped the design decisions during development of the process, can be summarised as follows (as defined in Section 2.8):

1. End-to-end support for the software development lifecycle.
2. Grounding in real-world agent design concepts.
3. Goal-based requirements engineering.
4. Modularity and process tailoring.
5. Practicality and detail.
6. Inclusion of a comprehensive support tool.

The remainder of this chapter will detail the NUMAP process, and demonstrate how it meets these goals.
4.3 Identifying Process Phases

The first step in developing the design process was to identify the stages of development that would be used within the process.

For the NUMAP process, a phased, iterative approach was chosen as the methodology for structuring the phases of development. The complexity of communications, and the need to precisely divide functionality between agents, means that a correctly documented design is a necessity in developing systems of any real sophistication. This does not preclude incremental development, and support for iteration is a core goal of NUMAP.

Decomposing the process into logical phases assists the developer in gradually stepping through the design process, moving steadily from initial requirements specification, through to high-level design, detailed design, and finally to implementation.

Examining existing processes shows that a variety of different phase decompositions have been used for segmenting the process. We consider three of the most popular processes, Tropos, Gaia and Prometheus below, for reference.

The Tropos methodology divides the process into four distinct phases: early requirements, late requirements, architectural design and detailed design [21]. The key point of interest in this decomposition is the splitting of requirements into two phases, with the early requirements phase being primarily focused on identifying actors within the system and their goals, while the late requirements phase is focused on viewing the entire system as an actor, and establishing its dependencies, and its functional and non-functional requirements.

The Gaia methodology [20] is decomposed into three distinct phases: requirements, analysis and design. Requirements capture is seen as being independent of the methodology, and is not specified. Analysis is the process of
gaining an understanding of the system and its structure, without considering implementation details – it is primarily concerned with viewing the system as an organisation, and identifying the roles within that organisation and their interactions. The design phase, as mentioned in Chapter 2, is quite high-level, primarily being concerned with identifying agent types, their services, and their interactions, or “acquaintances”.

The Prometheus methodology [22] is decomposed into three well-defined phases. The system specification phase is concerned with identifying the goals, inputs, outputs and basic functionalities of the system. The architectural design phase is concerned with determining the agent types for the system, and detailing their interactions. The detailed design phase specifies the internal structure of the agent.

From these three existing processes, it can be seen that a common decomposition would include, at the least, some form of requirements or high-level system specification phase, a phase focusing on the organisational view of the system, and a detailed implementation phase.

In defining our phases of the process, a smooth flow from start to finish is desirable, based on our goal of end-to-end development. Each phase of the process should smoothly transition into the next, using the information gathered in the previous phase as the basis for beginning the specification of the next.

An additional concern for the to-be-developed process is modularity. As modularity of the process is a core goal, the stages of development must be segmented in a manner that is conducive to allowing the process to be tailored by “swapping out” modules.

In order to aid in modularity, the requirements phase of the process should be as self-contained as possible. The requirements module to be developed for the initial process should ideally be based on an existing, well established goal-
based requirements analysis technique. Anton's GBRAM [111] has proven to be such a technique. Therefore, the initial phase of the process should be a requirements phase, taking advantage of this technique.

Once requirements are identified, a high-level view of the system can be established. This should be in a common format, regardless of which module is used for specifying requirements. Additionally, in line with the view that an agent is an autonomous software component that is "situated within and a part of an environment [and] senses that environment and acts on it" [37], we should also develop a high-level view of that environment. At this stage, the requirements should be elaborated upon, in an analysis phase, to develop an overview of the system, its agents, their high-level interactions, and how agents are situated in their environment.

Once this high-level analysis view of the system has been established, the interactions between agents can be specified in more detail, by elaborating how they are organised. This organisational design phase includes defining agent interactions in detail, specifying which services agents are to perform, how they are to communicate, and what their role is within the system.

Finally, a detailed design of each agent is required. Depending on which approach is to be used for implementing the agent, different modules could be used for this. It would even be conceivably possible to have different agents within the system use a different implementation technique. However, our focus for this agent design phase is to develop a module that will allow agents to be defined using the concepts commonly used for specifying BDI agents – in other words, using a PRS-derived [67] conceptual framework for designing agents.

Finally, based on the organisational and detailed design, agents can be implemented, in an implementation phase.
4.4 Overview of the NUMAP Process

This section introduces the NUMAP process itself, providing an overview of its main features.

4.4.1 Introducing NUMAP

NUMAP is a practical design process that guides the development of agent-based systems. It covers all aspects of design, from early requirements through to implementation. NUMAP provides a set of guidelines that define the essential concepts used in each phase of the design.

Rather than taking an abstract approach to defining the design concepts, NUMAP ensures that they have parallels in real-world agent implementation environments. In doing so, NUMAP allows software engineers to produce a design specification that is closer to the actual implementation, and which takes into account the specific requirements of the agent implementation technique that is being used.

In order to retain flexibility and support for different agent types, NUMAP uses a modular approach, where particular phases of the process can be replaced with a different module in order to support different design approaches.

For example, the current Agent Design and Implementation modules being used with NUMAP are based upon the BDI agent philosophy. The concepts defined during design are closely related to those used in BDI-based [11] implementation environments such as JACK, and Jadex. These could be easily swapped for Swarm based [134] agent design and implementation modules in order to support this different agent design approach. The NUMAP process has been designed to allow for such module changes without affecting the rest of the process.

Additionally, the design created by the process is itself modular. For example, if a decision is made to change agent implementation environments after the
design is complete, then only the parts of the system related to the agent implementation module need be changed. The rest of the design remains unchanged.

The key concepts that are defined in the process are goals and agent types. Agent types define the kinds of agents that can be instantiated, in the same way as classes define the objects that can be created in object-oriented software engineering. Goals define the objectives of the agent type.

NUMAP concepts are described by completing a series of forms. Each concept has its own distinct form, which lists the attributes that need to be defined to fully describe that concept.

The process itself is divided into five distinct phases (as seen in Fig. 4.1): Requirements Elicitation, Analysis, Organizational Design, Agent Design, and Implementation. In practice, software developers will iterate through the NUMAP process as they refine their software product.

![Figure 4.1: Overview an iteration of the NUMAP process](image)
Apart from analysis and organisational design, each of these phases can be altered or replaced in order to support a variety of design approaches. In particular, the Requirements phase can be altered to support different requirements elicitation methods, and the Agent Design phase can be replaced in order to support different agent design techniques.

4.4.2 Process Outline

The primary activity within each phase of NUMAP is specification of the concepts required for the system model. This is achieved using the previously mentioned forms.

Forms were chosen as the preferred mechanism for specifying design elements, because they allow developers to easily provide the required information for each concept via a series of fields. Forms can be readily implemented in a support tool to assist with the process, and are easily readable. They can be straightforwardly organised into an index.

Each form comprises a field-based list of the information required to define a concept in the system model, and these fields are filled in by the developer. Each field has a specific format for data, such as: free text, an identifier for another entity within the process, or semi-structured text.

A detailed system model is gradually constructed from the design elements described by the forms. Each phase of the process has its own distinct phase model, and together, they form the system model. As development progresses, the models from the previous phases are used to assist in constructing the model for the current phase. The NUMAP tool assists where possible in automating the transition between phases. The model for each phase consists of the elements defined for that phase via forms, and the relationships between them.
Diagrams can be generated from the system model to provide a visual representation of the model. They can also be used as an alternative mechanism for specifying details of the system, or for making minor alterations to the design.

A modular approach is used in NUMAP to aid with the generality of the process. This allows different requirements elicitation techniques and agent design approaches to be used, so the process is not tied to specific design philosophies.

Other processes, such as Gaia [20] omit specific design details in order to preserve generality. NUMAP uses a modular approach to allow for a more detailed design process, without restricting the entire design process to that approach. This combines the benefits of a more detailed design process with the advantages of generality.

Three phases of the process are modular, namely: requirements, agent design, and implementation. Due to the modular approach taken in the requirements phase, the initial part of the analysis phase differs slightly depending on which requirements approach is used, but its overall structure and outcomes are the same.

All of NUMAP’s modules are isolated from the rest of the process, so its core still retains generality. This also allows different design approaches to be taken within the same system. For example, some agents in a system may be BDI-based, while others may use different agent implementation techniques. This can be achieved by using different Agent Design modules for different agents within the same system.

Organisational design is at the core of NUMAP. It defines interactions between agents, and is based on FIPA [86] standards.

The requirements phase is a module, allowing for a number of requirements elicitation approaches to be used, including GBRAM [111] and KAOS [110]. The
process does not mandate which is used, as one may be more appropriate than another in different circumstances.

The requirements module may produce different outputs, depending on the elicitation technique used. These outputs are then transformed into a common format during the early stages of the analysis phase. At the end of this step, a series of well-defined outputs are produced.

The modular agent design phase allows for different agent techniques to be used, provided the agents’ external interfaces are the same.

The implementation phase allows a variety of implementation environments to be used. There are several different implementation platforms for BDI agents alone, such as JACK and Jadex. Implementation modules have been defined for both these support environments.

The design and implementation phases are isolated from the rest of NUMAP, because they deal with individual functionality of agents, which is self-contained for each agent.

NUMAP’s support tool provides a form-based GUI for entering the data for each phase of the process. Additionally, it enforces data integrity, provides assistance with transitioning between each of the phases of the process, and guides the developer through each step of the process.

The support tool assists with validating the correctness of the system model; for example, ensuring the pre-conditions and post-conditions for goals exist, ensuring that two goals are not preconditions of each other, and ensuring all goals have plans associated with them.

The tool also assists with generating diagrams from forms and vice-versa. These diagrams help visualise the system model, and can also be edited to directly update the system model.
The support tool is also used in the implementation stage to generate code for a specific agent implementation platform. The support tool can generate the basic code for an agent, producing a template, and defining the agent’s overall structure. The programmer uses this as a basis for implementing the agent. The code that is generated is for a specific implementation environment.

4.5 Process Phases

In Section 4.3, the phases of the NUMAP process were identified. A more detailed description of each phase of the process is provided in this section. The objectives of the phase are described, and the main concepts for each phase are explained. In subsequent sections, each phase will be defined in detail.

In NUMAP, the individual elements of each process are referred to as concepts. The formal specification of the process involves defining these “concepts” (e.g. “goals” or “agent types”) for each phase, how they interrelate, the data to be used for specifying them, and how they are to be used within the process. Throughout this section, the concepts are specified in italics.

4.5.1 Requirements

The Requirements phase uses a goal-based requirements analysis technique to describe the requirements for the system. The overall goals of the system are defined, along with an overview of how these goals will be achieved.

Any goal-based requirements method may be used at this stage. The requirements method currently being used with NUMAP is GBRAM [111], however this could be replaced with another approach by using a different Requirements module. Depending on the approach used, the specific concepts defined in this phase would differ.

The GBRAM-based requirements stage requires three distinct sets of data to be defined, as specified in the GBRAM process.
Firstly, the *agents and stakeholders* within the system need to be defined. Next, the *goals* for the system are defined. The GBRAM requirements elicitation method provides a number of strategies for defining these goals. These strategies are followed to create a detailed model of the high-level goal hierarchy of the system. Lastly, the operationalisations, or actions, of each goal are defined. These provide a basic description for how each goal may be achieved.

### 4.5.2 Analysis

The Analysis phase is concerned with creating an abstract model of the system based upon the results of the requirements phase.

There are two main goals of this phase. Firstly, the outputs of the requirements phase are mapped into a standard format. This is necessary due to the modular nature of the requirements process. Secondly, these design elements are expanded with additional detail, and a number of additional high-level properties are defined.

In order to complete the first task in this phase, the outputs from the requirements phase are re-specified in terms of analysis concepts. The primary aim of this step is to define the *goals* and *agent types* for the system. *Analysis plans* may also optionally be defined at this stage. These correlate with actions in GBRAM.

Once this mapping is complete, additional concepts are then defined.

The first of these is the system's *environment*. In order to define this, relevant elements that are external to the system are identified. These are elements that will be sensed or changed by agents within the system.

Secondly, agent *sensors* are defined. These are the mechanisms that allow an agent to sense environmental elements and update its internal beliefs based upon this information.
Thirdly, agent *effectors* are defined; these are mechanisms through which an agent effects change upon its environment.

Fourthly, *services* are specified that define the functionality that an agent type makes available to its peers. Services are an important concept within NUMAP, as they are used to model interactions between agents.

Services may be grouped into *roles*. Roles are cohesive groupings of agent services that can be applied to agent types that share specific functionality.

Agent types can be grouped into a number of *organisations*. This is necessary to divide the system into components.

At the conclusion of this phase, all of the above essential elements will be defined, ready for the organisational design phase.

### 4.5.3 Organisational Design

The next stage of NUMAP is Organisational Design. This phase is concerned with defining the system at an inter-agent level. The interactions between agent types are defined, along with the services provided by each agent type. The internal functionality for each agent is not defined at this stage.

During this stage, the concepts that were defined in the analysis phase are refined, and several additional concepts are added to fully specify inter-agent communication.

An organisational-level description of *agent types* is created, providing more detail regarding the simple types that were identified during the Requirements and Analysis phases. This forms the final list of agent types that will be used in implementation.
In order to ensure the cohesiveness of agent types at this stage, the agent types that were identified in the analysis stage may need to be divided and reclassified, and new types may need to be created.

*High-level goals* for each of these agent types are also defined at this stage. These are used for documentation purposes only, and should not be confused with the detailed goals that are defined in the agent design phase, which follows.

In order for agents to communicate, *message types* need to be defined. These specify the structure of messages that are used by agents to communicate. In order to assist with standards compliance, each message type is assigned a “performativa”, based on those defined in the FIPA Communicative Act Library Specification [131].

Agent communication is further structured by *agent interaction protocols*. These define a formal means that achieves the exchange of messages between two parties. The interaction protocols that are defined may be based on existing FIPA interaction protocol standards, for example Request Interaction Protocol [133], Query Interaction Protocol [135], or Contract Net Interaction Protocol [77].

*Services* are another aspect of agent communications. Agents may use the services of another agent, or may provide a number of services themselves.

As in the analysis phase, *roles* are groupings of related services. Each agent type has one or more roles, which indicates the services that the agent provides, and also the services that it uses.

*Organisations* are also carried over from the analysis phase, providing a mechanism for grouping similar agent types. As the agent types are altered during this phase, the organisations need to be adjusted to match.
Organisations can also be used as a basis for defining more complex structures for the system. For example, extensions to the organisational design phase that allows complex behaviour such as rules, norms and agent admission criteria will be explored in future work.

4.5.4 Agent Design

Agent design is a modular stage of NUMAP that involves defining the internal behaviour of each agent. The module used depends upon the approach that is preferred by the developer.

The initial implementation of NUMAP uses a BDI module for creating agents. This allows creation of agents that work with two of the most popular agent runtime environments, JACK, and Jadex.

Both of these environments use the BDI model of agents, along with a runtime environment that shares a large number of features with the PRS system [67]. NUMAP’s agent design module was designed with the PRS model in mind, but is applicable to all forms of BDI agents.

The primary concept to be modelled within the agent design phase is the agent type. Within the agent design phase, this defines the internal aspects of the agent.

All agent types have formal goals that define their proactive behaviour. An agent will generally have several goals that it simultaneously pursues. An agent attempts to satisfy its goals via plans. Each plan defines an action, or set of actions an agent may perform in pursuit of a goal or in response to a reaction.

Agent reactions are instant responses to events. Unlike goals, these do not run constantly, rather they model the reactive behaviour of the agent type. Events are triggered by agents in response to some situation that requires immediate attention. They can be generated by a received message, a belief state change, or they can be manually generated by a plan.
Beliefs define what an agent “thinks it knows”. They are the agent’s information about its environment and about other agents within the system. That is, beliefs are the agent’s symbolic representation of its surroundings.

The agent’s environment defines the elements external to the system with which the agent will interact. This includes environmental elements that will be sensed, as well as elements that will be changed by the agent. Sensors and effectors are the agent’s mechanisms for interacting with its environment. Sensors are used to receive inputs from the environment, while effectors are used by the agent to effect change upon its environment.

An agent uses plan selection rules to select which plan is used in an attempt to satisfy a goal. Similarly, when an agent must use a service, it may have to select from a number of agents that provide that service. Delegation selection rules are used to make that selection.

Agent capabilities can be used to create a grouping of beliefs, goals, reactions and plans that can be used by any agent. Specific agent functionality can be defined in the capability, and inherited by an agent type, thus encouraging software reuse.
4.5.5 Implementation

Implementation involves writing code for the finished design, which will be executed in an agent runtime environment.

The close mapping between the concepts defined in the design phases and the actual implementation environments allows for agents to be readily implemented from the design specification. To further aid this process, the basic structure of the agent code is automatically generated by the NUMAP support tool. This will be described further in Chapter 5.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Concepts Defined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>Agent/Stakeholder</td>
</tr>
<tr>
<td></td>
<td>Goals</td>
</tr>
<tr>
<td></td>
<td>Actions</td>
</tr>
<tr>
<td>Analysis</td>
<td>Environment Element</td>
</tr>
<tr>
<td></td>
<td>Agent Types</td>
</tr>
<tr>
<td></td>
<td>Agent Goals</td>
</tr>
<tr>
<td></td>
<td>Organisations</td>
</tr>
<tr>
<td></td>
<td>Roles</td>
</tr>
<tr>
<td></td>
<td>Sensors</td>
</tr>
<tr>
<td></td>
<td>Effectors</td>
</tr>
<tr>
<td></td>
<td>Services</td>
</tr>
<tr>
<td></td>
<td>Analysis Plans</td>
</tr>
<tr>
<td>Organisational</td>
<td>Agent Types</td>
</tr>
<tr>
<td>Design</td>
<td>Message</td>
</tr>
<tr>
<td></td>
<td>Role</td>
</tr>
<tr>
<td></td>
<td>Service</td>
</tr>
<tr>
<td></td>
<td>High-Level Goal</td>
</tr>
<tr>
<td></td>
<td>Interaction Protocol</td>
</tr>
<tr>
<td></td>
<td>Environment Element</td>
</tr>
<tr>
<td></td>
<td>Organisation</td>
</tr>
<tr>
<td>Agent Design</td>
<td>Agent Types</td>
</tr>
<tr>
<td></td>
<td>Plan</td>
</tr>
<tr>
<td></td>
<td>Goal</td>
</tr>
<tr>
<td></td>
<td>Reactions</td>
</tr>
<tr>
<td></td>
<td>Plan Selection Rules</td>
</tr>
<tr>
<td></td>
<td>Delegation Selection Rules</td>
</tr>
<tr>
<td></td>
<td>Sensors</td>
</tr>
<tr>
<td></td>
<td>Effectors</td>
</tr>
<tr>
<td></td>
<td>Beliefs</td>
</tr>
<tr>
<td></td>
<td>Capabilities</td>
</tr>
</tbody>
</table>

Table 4.1. A summary of the concepts defined at each stage of the NUMAP process
Once implementation and testing are undertaken, the software will be assessed, and any additional requirements identified. The process may then begin again, in a new iteration, addressing the new requirements that have been identified.

4.5.6 Summary

This section provided a high-level overview of the phases of the NUMAP process, their purposes, and the concepts used in specifying each phase. This is summarised in Table 4.1.

The remainder of Chapter 4 will describe the phases of the process in detail, specifying the following information for each phase:

- Its purpose.
- How its concepts are specified.
- Why each concept is necessary.
- How transitions between phases are achieved.

4.6 Requirements Phase

4.6.1 Purpose of the Requirements Phase

The requirements elicitation phase is a modular phase of NUMAP, designed to be independent of the rest of the process. Any requirements elicitation technique may form the basis of a NUMAP requirements module. However a goal-based technique makes it easier to transition to the rest of the process, which is defined around the notions of agents and goals, so this approach is used at in the default process.

The requirements elicitation phase embodies the requirements gathering process, gathering domain information from stakeholders and setting out the requirements for the system.
4.6.2 Goal-Based Requirements Analysis Techniques

In selecting a requirements analysis approach to be used with the default NUMAP requirements module, two techniques were considered, KAOS [110], as used by the Tropos methodology, and the Goal Based Requirements Analysis Method (GBRAM) [111].

4.6.2.1 KAOS

KAOS [110] is a goal-oriented requirements framework that provides a technique for specifying requirements using goal-oriented concepts not covered by existing requirements capture techniques. These concepts include goals, agents, and alternatives.

KAOS is intended to solve two main deficiencies with existing formal specification languages for requirements. These deficiencies are: the limitation of only dealing with functional requirements, due to a restricted set of built-in abstractions; and the inability to address the issue of actually acquiring the requirements in the first place.

In order to address these deficiencies, KAOS divides the requirements gathering process into two main tasks: requirements acquisition and formal specification.

KAOS has three components: a conceptual model for requirements, with an associated acquisition language; strategies of elaborating requirements models; and an automated assistant to assist with the acquisition process. These components have been designed with the intention of maximising the use of domain knowledge during the requirements gathering process.

The approach operates at three levels. At the meta-level, basic concepts such as agents, actions and links are defined. At the domain level, concepts relevant to the application domain are defined, being instances of meta-level abstractions. For example, "Author" may be an instance of "Agent", and "Publish" may be an
instance of the "Action" meta-concept. The instance level consists of instances of the domain-level concepts.

A goal in KAOS is seen as "a nonoperational objective to be achieved by the composite system". That is, goals can not be achieved by the actions of a single agent in the system. KAOS goals are divided according to five basic patterns: achieve, cease, maintain, avoid and optimise.

As well as these patterns, goals may be divided into different categories. SystemGoals are goals that must be achieved by the system as a whole. PrivateGoals are goals that may be achieved by an individual agent. Additionally, goals may be of a variety of types, including SatisfactionGoals, InformationGoals, ConsistencyGoals, SafetyGoals, and PrivacyGoals.

During the requirements phase, abstract goals are refined into more concrete ones, and then often divided into subgoals. The Reduction meta-relationship can be used to describe the different combinations of subgoals that can be used to achieve the goal.

Another fundamental concept of the KAOS method, constraints, represent operational objectives of the system (i.e. they can be satisfied by the actions of a single agent in the system), and are expressed in terms of objects and actions available to the system. Constraints may represent, for example, some condition that needs to be maintained.

Goals can be made operational through constraints, if the goal can be met by adhering to certain constraints. Once goals are “operationalised”, they can be readily implemented. A goal that is operationalised can not be further reduced. The "Ensuring" and "Responsibility" relationships can be used to guarantee that an agents' actions do not violate the constraint. KAOS also provides other relationships, such as Structuring and Composition, which can be used to structure the system.
The requirements gathered are expressed in an acquisition language, which is based upon the structure of the meta-model. A requirements model is regarded as an instance of this meta-model, represented as a graph, with abstractions as nodes and links as edges. Strategies may be defined for the acquisition process, consisting of a number of steps, and "define specific ways of traversing the meta-model graph to acquire instances of its various nodes and links". Domain models are represented in the same language as requirements.

The suitability of KAOS to an agent-based system is somewhat reduced by its refinement of goals into actions. This task is best performed at design-time or even run-time with an agent system, allowing the full capabilities of plan selection and reasoning within agent systems to be taken into account.

4.6.2.2 GBRAM

The Goal Based Requirements Analysis Method (GBRAM) [111] is a requirements elicitation approach that is used for the identification and organisation of goals for requirements analysis, and their operationalisation into high-level actions. Rather than simply focusing on the features the system is to provide, goal-based requirements analysis has the motivation for the design of the system as its primary concern.

In developing GBRAM, the designers identified two main issues in goal-based requirements engineering: identifying the system’s goals (goal analysis) and managing changes in the goals (goal evolution). The initial step in GBRAM is goal analysis, that is determining the goals from existing organisational documentation and process descriptions. Several techniques are used for this, including scenario analysis; identification of goal obstacles and constraints; and goal operationalisation. Goal evolution is also a necessary consideration. While according to the authors, goals generally change much more slowly than processes and other abstractions, it is necessary to refine goals after their initial identification. This will be further described below.
The main abstractions used in the GBRAM process are: goals, requirements, operationalisation, agents, constraints, goal decomposition, scenarios, and goal obstacles. Goals represent the aims of the system, and describe the reasons why the system is needed. There are two kinds of goal, achievement goals and maintenance goals. Achievement goals describe some objective to be achieved, while maintenance goals are reliant on maintaining (or avoiding) a certain condition. Goals are further specified by Requirements, which state how the goal is to be accomplished. In order for goals to be fully defined, Goal Decomposition needs to take place, i.e. the goals need to be subdivided into more specific goals. As with KAOS, goals are operationalised by ensuring that their subgoals are sufficiently defined to ensure that they have an operational definition, i.e. that they can be effectively implemented. In GBRAM, agents are assigned responsibility for achieving goals within the system. Constraints describe the conditions that need to achieved in order for the goal to be deemed achieved. Goal obstacles and Scenarios are used to describe situations that may lead to the failure of a goal. Obstacles denote reasons why a goal may fail, while scenarios define the specific circumstances under which this failure may occur.

Goal analysis primarily consists of identifying goals from documentation about the existing organisational processes and any specifications regarding the system that is to be implemented. These documents generally describe information in terms of actions and workflows rather than goals, but goals may be derived from these documents. As goals are identified, the agents that are responsible for them are also defined, and the constraints that specify when the goal is accomplished are identified.

The goals that are initially identified will also change, both during the initial analysis phase as more detail is obtained, and over the life-cycle of the project as goals and priorities are altered. Goal evolution can be by elaboration, i.e. the identification of obstacles and their scenarios, and by refinement, i.e. merging goals to reduce redundancy, identifying further constraints, and
operationalising goals. Additionally, achievement goals are prioritised based on dependencies.

Operationalisation of goals takes place by creating a set of goal schemas for the goals that have been identified. Goals are related to agents by events that cause state transitions, and goals and their actions are specified by preconditions/assumptions (the required state at goal initiation) and postconditions/results (the state of the system after the goal is achieved).

4.6.2.3 Choice of Requirements Technique

The means of requirements specification selected for use in NUMAP was GBRAM. The technique maps very closely to the concepts used in specifying BDI systems, and it provides a mature and straightforward approach for specifying goals.

This approach centres on determining and specifying what goals the system should be designed to achieve through the process of goal analysis, that is, identifying the goals from documentation, specifications and stakeholder interviews, and managing goal evolution as more detail is obtained.

The result of applying the GBRAM technique is an output comprised of three parts: a list of stakeholders, or agents, who are involved in, or influenced by the system; a list of goals that must be achieved by the system; and a list of goal operationalisations, or actions that are taken by relevant stakeholders to achieve the goals for which they are responsible.

4.6.3 Specification of Requirements Phase Concepts

In the GBRAM requirements module, several concepts are defined to represent these outputs of the GBRAM process. These represent the outputs of the GBRAM technique in its original form.

The following concepts will be defined:
• **Agents/Stakeholders**
• **Goals**
• **Actions**

The syntax used here, and for the remainder of the chapter is as follows:

• An identifier for the concept is first provided (e.g. R1).
• A description of the concept and its properties is provided.
• The concept is presented in a table. These tables are based upon the “Standardised Properties” tables commonly used in OMG [16] specifications.

### 4.6.3.1 Agent/Stakeholder (R1)

This concept represents the agents and stakeholders identified within the GBRAM process. Stakeholders in GBRAM are any actor that is involved in, or influenced by the system - in other words, any actor with a stake in the outcome of a goal. In contrast, agents are direct actors that pursue goals within the system. It is entirely possible for a particular entity to be an agent for a particular goal, and a stakeholder in another.

The Agent/Stakeholder concept has the following properties:

- **Name**: This is the name used to identify the agent or stakeholder. It should be a brief, descriptive name denoting the title of the entity or the role it plays within the system.

- **Type**: This property may have one of three values. “Agent” indicating that the entity represented is an agent within the system, that is, it takes an active role in the system and is responsible for achieving some goal(s). “Stakeholder” indicates that the entity represented is affected by the outcomes of a goal of the system, but is not directly responsible for achieving that goal. An entity marked solely as “Stakeholder” will not be responsible for achieving any goals in the system. “Agent/Stakeholder” indicates that the entity plays the role of both
Agent and Stakeholder, being responsible for achieving some goals, while also being affected by the outcomes of other goals. In reality, most agents will take the form of an Agent/Stakeholder, as any entity responsible for achieving a goal will have a stake in its fulfilment.

**Description:** A brief description of the agent or stakeholder, indicating its importance within the system.

*Table 4.2* presents these properties. It is based upon the “Standardised Properties” tables commonly used in OMG [136] specifications.

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Customer’</td>
<td>Name of agent or stakeholder</td>
</tr>
<tr>
<td>Type</td>
<td>Agent</td>
<td>Stakeholder</td>
<td>Agent/Stakeholder</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Buys goods ...’</td>
<td>A brief description of the agent or stakeholder</td>
</tr>
</tbody>
</table>

**Table 4.2 – Agent/Stakeholder (R1)**

### 4.6.3.2 Goal (R2)

This represents the goals to be achieved within the system. Goals are the reasons for the system’s existence, and in essence, describe why the system is needed, and what it is to achieve. There are two kinds of goals: *achievement goals* and *maintenance goals*. Goals are associated with a set of conditions, scenarios, preconditions, and postconditions. Goals may be grouped hierarchically – in line with GBRAM’s strategy of goal decomposition.

Goals have the following properties (specified in *Table 4.3*):

**Name:** This is the name used to identify the goal. It should be a brief, descriptive name denoting what the goal is trying to achieve.
**Type:** This property may have one of two values. “Achievement” indicates that the goal is a GBRAM achievement goal, i.e. some specific objective that is to be achieved. “Maintenance” indicates that the goal is a GBRAM maintenance goal, i.e. some condition that is to be kept within certain bounds.

**Description:** A brief description of the goal, indicating its importance within the system.

**Actions:** A collection of actions that may be used to achieve, or “operationalise” the goal. These actions are further defined below. There may be more than one alternative action for achieving a goal.

**Agent:** The agent responsible for achieving the goal.

**Stakeholder:** A list of stakeholders with an interest in the goal. These are any entities that have a direct stake in the fulfilment of the goal. The agent responsible for the goal will also usually be listed as a stakeholder.

**Obstacles:** These represent behaviours or other goals that can prevent this goal from being achieved. These provide a list of ways in which a goal may fail, and are useful for assisting in identifying exceptional cases for functionality. They are used as the basis for creating scenarios, which are to be written as a textual description. A goal may have several scenarios.

**Scenarios:** These represent specific situations under which a goal may fail. Based on the concept of scenario analysis [137], they indicate situations which, if encountered, will result in goal failure. Unlike Obstacles, they identify specific restricted conditions, not general situations. Scenarios are to be written as a textual description, a goal may have several scenarios.

**Constraints:** These are identified during the requirements process, and are descriptions of the situations that must be met for a goal to be achieved. During the GBRAM process, these will be refined into pre-conditions and post-conditions for the goal, but the facility is provided in the NUMAP process to
record these higher-level representations, to allow them to be recorded during the requirements elicitation process. As in GBRAM, these are to be written as a textual description. A goal may have several constraints.

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Account created’</td>
<td>Name of the goal.</td>
</tr>
<tr>
<td>Type</td>
<td>Achievement</td>
<td>Maintenance</td>
<td>‘Agent’</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Create account …’</td>
<td>A brief description of the goal.</td>
</tr>
<tr>
<td>Actions</td>
<td>Action(R3)[]</td>
<td>{AccountAddition, AccountPorting}</td>
<td>A collection of actions that can be used to achieve the goal.</td>
</tr>
<tr>
<td>Agent</td>
<td>Agent/Stakeholder(R1)</td>
<td>Manager</td>
<td>The agent responsible for achieving the goal.</td>
</tr>
<tr>
<td>Stakeholder</td>
<td>Agent/Stakeholder(R1)[]</td>
<td>{Employee, Manager}</td>
<td>The agents/stakeholders with a direct interest in the outcome of the goal.</td>
</tr>
<tr>
<td>Obstacles</td>
<td>String[]</td>
<td>‘Name clash’</td>
<td>Ways in which a goal may fail.</td>
</tr>
<tr>
<td>Scenarios</td>
<td>String[]</td>
<td>‘Username exists’</td>
<td>Specific situations that will cause a goal to fail.</td>
</tr>
<tr>
<td>Constraints</td>
<td>String[]</td>
<td>‘Account has been created’</td>
<td>Conditions that must be met for the goal to be achieved.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>String[]</td>
<td>‘System online’</td>
<td>Situations that must have arisen before the goal is attempted to be achieved.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>String[]</td>
<td>‘Account processed’</td>
<td>Specific conditions that must be met for the goal to be achieved.</td>
</tr>
<tr>
<td>Parent</td>
<td>Goal(R2)</td>
<td>Handle Accounts</td>
<td>The parent goal in the goal hierarchy.</td>
</tr>
</tbody>
</table>

Table 4.3 – Goal (R2)

*Preconditions:* These are a set of conditions that must be met before a goal can start being pursued. They reflect the necessary system state for an agent to begin an attempt to fulfil a goal. They (along with post-conditions) are an outcome of an analysis of the constraints that have been identified. Preconditions are to be written as a textual description. A goal may have several preconditions.

*Postconditions:* These are the conditions that must be met for a goal to be considered to be achieved. For achievement goals, these are the requirements
that must be met before a goal can be considered to be completed. For maintenance goals, they are the required state to be maintained, or a state to be avoided. Postconditions are to be written as a textual description. A goal may have several postconditions.

*Parent:* Goals in GBRAM may be organised into a hierarchy. The parent goal of an agent must therefore be specified as part of the goal.

### 4.6.3.3 Action (R3)

This represents a behaviour that will be used to achieve a goal. Collectively, these are the actions the system takes to meet its objectives.

Actions have the following properties (specified in *Table 4.4*):

*Name:* This is the name used to identify the action. It should be a brief, descriptive name denoting what task the action performs.

*Description:* A brief description of the action, indicating its importance within the system.

*Reads:* This is the data to be read by the action, i.e. what data it requires in order to perform its functionality. An action may require more than one piece of data. The data required is recorded in text form.

*Changes:* This is the data the action will change during its execution. Any data that may be changed in the system during the performance of the action is listed. An action may change more than one piece of data. The data to be changed is recorded in text form.

*Assumes:* This shows the state the environment must be in before an action is performed. These state conditions are generally derived from an associated goal’s preconditions and scenarios. These are to be written as a textual description. An action may have several assumptions.
Results: This is a description of the end result to be achieved by the action. This is a brief textual description of the outcome that will be achieved by successful performance of the action. An action may have several results.

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Create account’</td>
<td>Name of the action.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Create account …’</td>
<td>A brief description of the action.</td>
</tr>
<tr>
<td>Reads</td>
<td>String[]</td>
<td>{‘AccountDatabase’, ‘EmployeeDatabase’}</td>
<td>Data to be read by the action.</td>
</tr>
<tr>
<td>Changes</td>
<td>String[]</td>
<td>{‘AccountDatabase’}</td>
<td>Data to be changed by the action.</td>
</tr>
<tr>
<td>Assumes</td>
<td>String[]</td>
<td>{‘Databases Online’}</td>
<td>Conditions that need to be true for the action to proceed.</td>
</tr>
<tr>
<td>Results</td>
<td>String[]</td>
<td>{‘Account created’}</td>
<td>The result of the action</td>
</tr>
</tbody>
</table>

Table 4.4 – Action (R3)

4.6.3.4 Mapping the Requirements Phase to the Analysis Phase

At the conclusion of each phase of the process, the information gathered during the phase is used to begin populating information for the subsequent phase, acting as a start-point for the next phase. This task is aided by the NUMAP support tool.

These mappings are presented in Appendix B, as the tables involved are quite large.

4.6.3.5 Summary

The above descriptions detail the information gathered during the GBRAM Requirements module of the process. At the conclusion of the phase, a full goal-oriented description of the requirements of the system will have been achieved.
4.7 Analysis Phase

4.7.1 Purpose of the Analysis Phase

The analysis phase of the process is concerned with creating an abstract model of the system based upon the results of the requirements phase. The results of the requirements phase are mapped into a standard format, which is required as part of NUMAP’s support for modular requirements phases.

Secondly, additional detail is provided, to form a more comprehensive high-level view of the system and its environment. This involves definition of the system’s environment, the roles played by agents, the services provided as part of that role, the sensors and effectors used by agents, and a preliminary organisational grouping of agents.

At the conclusion of the analysis phase, a high-level view of how the system will function will have been established.

4.7.2 Concepts Defined in the Analysis Phase

If a goal-based requirements analysis method was used for the requirement module, then goals, agents and actions would have been specified prior to beginning the analysis process. If another requirements technique was used, the first task for the analysis phase should be transforming those requirements into a goal-based structure.

In addition to these concepts, we add a number of new elements to our analysis. Environmental aspects are defined, roles and services are specified, and basic organisational groupings of agents are established.

4.7.2.1 Defining the Environmental Concepts for Analysis

Most agents are situated in, and interact with, some form of environment. In industrial applications, that may be a physical environment, in which changes in
the environment are detected by sensors and changes made to the environment by means of “effectors”. Or, in a data-centric application, the environment may be a set of data.

Regardless, the elements of the system outside the agents themselves play an important part in the design of multiagent systems [138]. In fact, according to some authors, consideration of environment is part of what makes agents unique, and is a key part of their identity. According to Franklin and Graesser: “Change the environment and we may no longer have an agent. A robot with only visual sensors in an environment without light is not an agent” [37].

As the environment of an agent is key to our understanding of its functionality, and even its identity, it is necessary that the environmental aspects of the system are discovered and documented during the high-level analysis of the system.

Existing agent methodologies specify agent environments to varying extents. The Gaia methodology [20] does not specify the environment itself as a standalone concept at the analysis stage. The environment surrounding the agent is only specified in terms of the roles to be played by agents – it is purely considered input and output data to internal agent processes. An extension to the Gaia methodology provides for improved modelling of the environment, treating it as a primary abstraction [80].

The Prometheus process [22] considers environmental aspects of the system to a greater extent. During the System Specification phase, data external to the agent system is modelled, along with its inputs and outputs to the environment. These are termed *data, percepts and actions*.

In the Tropos methodology, the environment is considered in terms of actors. This primarily occurs in the Late Requirements phase, where the system as a whole is modelled as an actor [21]. The environment is therefore not explicitly
modelled as a unique concept – rather, it is expressed in terms of actors and their beliefs.

The NUMAP process defines attributes relating to the environment as standalone concepts, emphasising how necessary they are to the overall functionality of the system. Any information that exists in the agent’s environment is termed an environment element; these may represent physical attributes of the system, or data with which an agent interacts. Agents sense this data through sensors, and alter this data through effectors, using the terminology that is traditionally used in physical agent design [139].

These three concepts, environment elements, sensors and effectors form the basis of the NUMAP modelling for agent environments.

4.7.2.2 Defining the Role and Service Concepts for Analysis

The modelling of roles within agent systems is important, to adequately describe the position and responsibilities of an agent within a system [140]. Agent responsibilities are centred on the services they provide to other agents within the system. In their role, they will also use services provided by other agents.

These concepts have been handled in a number of ways by other processes. In Tropos, “strategic dependencies” are used as a way of describing the commitments between agents [21], and the tasks that an agent may perform on behalf of another agent.

In Gaia, these concepts are described by modelling the roles and responsibilities that an agent may have in the system [20]. Each agent’s expected functionality is described through a “roles model”, while the functionality of an agent is defined through its “responsibilities”.

115
In the Prometheus methodology, “scenarios” are defined during the system specification phase, to describe how tasks will be performed in the system, but the actual division of these tasks to individual agents is deferred to the Architectural Design phase [22].

As an agent’s roles and responsibilities play a critical part in describing and understanding how it fits into the overall system, these concepts are modelled at the NUMAP analysis phase. Firstly, the services an agent may perform are identified, and then the set of services an agent uses or provides are aggregated into roles. There is generally not a one-to-one mapping between agents and roles. Multiple agents may perform the same role, and multiple roles may be performed by a single agent.

The services provided and used by agents therefore form the basis for modelling the roles in the system.

**4.7.2.3 Defining the Organisational Concepts for Analysis**

There is a number of approaches to defining organisations within multiagent systems. An extension to the Gaia methodology proposes structuring the entire system around a series of organisations with their own roles, organisational rules and organisational structures [80]. Dignum and Weigand propose that multiagent systems should be structured in a manner resembling typical business organisational approaches [81].

However, there is little consensus about exactly how agent organisations should be structured, how rules and norms should be implemented, and whether they should be dynamically assembled at runtime, or be static in their nature. There is also no direct support within leading agent implementation platforms for organisational concepts. Defining organisational frameworks for structuring agent systems is in itself an emerging area of research.
As such, NUMAP takes a basic approach to defining organisations, viewing them primarily as a mechanism for structuring the system into logical components. In complex systems, it is likely that there will be groupings of agents that interact with each other to achieve common goals, in a large part separately from other agents in the system. Defining organisations at the analysis phase allows for componentisation of the system, aiding in ensuring its elements are as cohesive as possible.

Therefore, at the NUMAP analysis phase, an additional concept is introduced, the *organisation*.

### 4.7.3 Specification of Analysis Phase Concepts

The analysis phase of the NUMAP process is largely concerned with creating an abstract model of the system based upon the results of the requirements phase. The results of the requirements phase are mapped into a standard format, which is required as part of NUMAP’s support for modular requirements phases.

A number of additional concepts are added, as introduced above. These help to form a more comprehensive high-level view of the system and its environment.

The following concepts are defined for the analysis phase:

- *Environment Element*
- *Agent Types*
- *Agent Goals*
- *Organisations*
- *Roles*
- *Sensors*
- *Effectors*
- *Services*
- *Analysis Plans*
At the conclusion of the Analysis phase, a high-level view of the functioning of the system will have been established.

4.7.3.1 Environment Element (A1)

This concept models the relevant types of data that exist in the environment of the system. These elements consist of anything that will be read or changed by agents in the system, but does not include data that is internal to agents (agent beliefs). Environment elements are necessary to allow developers to specify how agents will interact with their environment. They may contain information that is gathered from the user.

Environment elements have the following properties (shown in Table 4.5):

**Name:** This is the name used to identify the environment element. It should be a brief, descriptive name denoting what the element is, or what data it contains.

**Description:** A brief description of the environment element, indicating its importance within the system.

**Attributes:** A list of the attributes of the environment element, listing the properties that are contained within the element. These are defined in a similar manner to specifying the attributes of a business object in object-oriented design, and are provided in text format.

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Accounts Database Record’</td>
<td>Name of the element.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Stores account information …’</td>
<td>A brief description of the element.</td>
</tr>
<tr>
<td>Attributes</td>
<td>String[]</td>
<td>{‘Name’,’Balance’}</td>
<td>Attributes of the element.</td>
</tr>
</tbody>
</table>

Table 4.5 – Environment Element (A1)
Parent: Environment elements may be nested into a hierarchy (e.g. a “machine” element may contain sub-elements for the various components of the machine). This property indicates the parent (more general) element of the one that is being specified.

4.7.3.2 Agent Type (A2)

This concept represents the agents and stakeholders within the system. Agent Types include additional information on top of what may have been identified in the GBRAM requirements elicitation process. Sensors and Effectors for the agent are listed, and the roles played by the agent are defined.

In contrast to GBRAM, in the analysis phase goals relevant to an agent are listed in the agent specification (rather than agents being listed as part of goals). This allows different agents to share the same goals, and to better match the structures used when implementing agents on implementation platforms such as Jadex [39].

During the transition between the Requirements Phase and the Analysis phase, these relationships between goals and agents will need to be transposed.

The Agent Type concept has the following properties (presented in Table 4.6):

Name: This is the name used to identify the agent. It should be a brief, descriptive name denoting the title of the entity or the role it plays within the system.

Type: This property is as specified in the GBRAM requirements phase. As in the specification of agents in the requirements phase, it indicates whether the agent is a stakeholder or a proactive agent.

Description: A brief description of the agent or stakeholder, indicating its importance within the system.
**Sensors:** A collection of “sensors” that are used by the agent to obtain information from its environment. The concept of sensors is further defined below.

**Effectors:** A collection of “effectors” that are used by the agent to alter its environment. The concept of effectors is further defined below.

**Plans:** A collection of actions that are used by the agent to achieve its goals. The collection of plans available to an agent is referred to as its “plan library”. The concept of plans is further defined below.

**Roles:** The roles played by the agent. Indicates what responsibilities the agent agrees to take, and therefore which services it will provide. The concept of roles is further defined below.

### Key Domain Example Description

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Customer’</td>
<td>Name of agent or stakeholder</td>
</tr>
<tr>
<td>Type</td>
<td>Agent</td>
<td>Stakeholder</td>
<td>Agent/Stakeholder</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Buys goods …’</td>
<td>A brief description of the agent or stakeholder.</td>
</tr>
<tr>
<td>Sensors</td>
<td>Sensor(A6)[]</td>
<td>{'Database Connector’, ‘User dialog’}</td>
<td>Sensors used by the agent to gather information from its environment.</td>
</tr>
<tr>
<td>Effectors</td>
<td>Effector(A7)[]</td>
<td>{'File writer’, ‘Motor controller’}</td>
<td>Effectors used by the agent to alter the state of its environment.</td>
</tr>
<tr>
<td>Plans</td>
<td>Plan(A9)[]</td>
<td>{'WriteFile’, ‘ReadData’}</td>
<td>Plans used by an agent to achieve its goals.</td>
</tr>
<tr>
<td>Roles</td>
<td>Role(A5)[]</td>
<td>{'Manage customers’, ‘Order products’}</td>
<td>The roles played by the agent, i.e. its core responsibilities to other agents.</td>
</tr>
<tr>
<td>Goals</td>
<td>Goal(A3)[]</td>
<td>{'User added’, ‘Stock replenished’}</td>
<td>The goals the agent is to achieve.</td>
</tr>
<tr>
<td>Stakeholder Goals</td>
<td>Goal(A3)[]</td>
<td>{'Order processed’, ‘Delivery processed’}</td>
<td>The goals the agent has a stake in the outcome of.</td>
</tr>
</tbody>
</table>

*Table 4.6 – Agent Type (A2)*
**Goals:** The goals that the agent will pursue. The collection of goals an agent is committed to will, in a large part, define its functionality and its place within the system. The concept of goals in this phase is further defined below.

*Stakeholder Goals:* Indicates in which goals this agent is a stakeholder. This is an important consideration, as it allows the developer to better manage the relationships and dependencies within the system. Changes to the functionality of a plan may well have an impact on all agents that are stakeholders in the goal that is satisfied by that plan.

### 4.7.3.3 Goal (A3)

This concept represents the goals to be achieved within the system. If the GBRAM approach was used, this is a refined version of the goals identified in that phase.

Goals in the analysis phase are defined in much the same way as during the requirements phase. However, during the analysis phase, goals identified during requirements may be refined or split, as the overall view of the system (and its environment) is developed.

If a requirements elicitation approach other than GBRAM is used, this phase also presents an opportunity to specify details (such as obstacles or constraints) that may not have yet been detailed.

This concept has the following properties (shown in *Table 4.7*):

**Name:** This is the name used to identify the goal. It should be a brief, descriptive name denoting what the goal is trying to achieve.

**Type:** This property may have one of two values. “Achievement” indicates that the goal is an achievement goal, that is, a goal that attempts to bring about some new state within the system. “Maintenance” indicates that the goal is a maintenance goal, preserving a pre-existing state within the system.
Description: A brief description of the goal, indicating its importance within the system.

Obstacles: As with the requirements phase, these represent behaviours or other goals that can prevent this goal from being achieved. These are to be written as a textual description. A goal may have several obstacles.

Scenarios: As with the requirements phase, these represent specific situations under which a goal may fail. These are to be written as a textual description. A goal may have several scenarios.

The difference between obstacles and scenarios is subtle. Obstacles denote reasons why a goal may fail, while scenarios define the specific circumstances under which this failure may occur.

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Account created’</td>
<td>Name of the goal.</td>
</tr>
<tr>
<td>Type</td>
<td>Achievement</td>
<td>Maintenance</td>
<td>‘Agent’</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Create account …’</td>
<td>A brief description of the goal.</td>
</tr>
<tr>
<td>Obstacles</td>
<td>String[]</td>
<td>{‘Name clash’}</td>
<td>Ways in which a goal may fail.</td>
</tr>
<tr>
<td>Scenarios</td>
<td>String[]</td>
<td>{‘Username exists’}</td>
<td>Specific situations that will cause a goal to fail.</td>
</tr>
<tr>
<td>Constraints</td>
<td>String[]</td>
<td>{‘Account has been created’}</td>
<td>Conditions that must be met for the goal to be achieved.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>String[]</td>
<td>{‘System online’}</td>
<td>Situations that must have arisen before the goal is attempted to be achieved.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>String[]</td>
<td>{‘Account processed’}</td>
<td>Specific conditions that must be met for the goal to be achieved.</td>
</tr>
<tr>
<td>Parent</td>
<td>Goal(A3)</td>
<td>Handle Accounts</td>
<td>The parent goal in the goal hierarchy.</td>
</tr>
</tbody>
</table>

Table 4.7 – Goal (A3)
Constraints: As with the requirements phase, these are identified during the requirements process, and are descriptions of the situations which must be met for a goal to be achieved. These are to be written as a textual description. A goal may have several constraints.

Preconditions: As with the requirements phase, these are a set of conditions that must be met before a goal can start being pursued. These are to be written as a textual description. A goal may have several preconditions.

Postconditions: As with the requirements phase, these are the conditions that must be met for a goal to be considered to be achieved. These are to be written as a textual description. A goal may have several postconditions.

Parent: Goals may be organised into a hierarchy. The parent goal of an agent must therefore be specified as part of the goal.

4.7.3.4 Organisation (A4)

This concept models logical groupings of agents. Organisations are used to componentise the system, grouping tightly-coupled agents, and allowing presentation of a higher-level organisational view of the system. An organisation has the following properties (shown in Table 4.8):

Name: This is a brief, descriptive name used to identify the organisation.

Description: A brief description of the organisation, indicating its importance within the system.

Interaction Organisations: This is a list of organisations with which this one interacts. That is, the organisations with which this organisation has a dependency. These interactions will generally be through communications with, or the use of a service from, an agent in the other organisation.
**Member Agents:** A list of the agents that are a member of the organisation. An agent may be a member of multiple organisations.

**Parent:** Organisations may be organised into a hierarchy. The parent organisation must therefore be specified, if it exists.

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Commerce Agents’</td>
<td>Name of the organisation.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Agents buying/selling goods …’</td>
<td>A brief description of the organisation.</td>
</tr>
<tr>
<td>Interaction</td>
<td>Organisation(A4)[]</td>
<td>{Courier Agents}</td>
<td>The organisations this one interacts with.</td>
</tr>
<tr>
<td>Member Agents</td>
<td>Agent(A2)[]</td>
<td>{Supplier, Retailer, Customer}</td>
<td>The agent types that are a member of the organisation.</td>
</tr>
<tr>
<td>Parent</td>
<td>Organisation(A4)</td>
<td>{Simulation Agents}</td>
<td>The parent organisation.</td>
</tr>
</tbody>
</table>

**Table 4.8 – Organisation (A4)**

### 4.7.3.5  Role (A5)

This concept models roles that may be played by agents. Roles encapsulate individual services, and represent responsibilities to which an agent commits. An agent that adopts a role indicates that it will provide all of the services contained within that role. In this way, a role represents both a commitment by an agent that adopts it and acts a logical grouping for services.

A role has the following properties (shown in Table 4.9):

**Name:** This is a brief, descriptive name used to identify the role.

**Description:** A brief description of the role, indicating its importance within the system.
Provided Services: These are the services that an agent adopting this role commits to providing. More detail on services is provided below.

Used Services: These are the services that will necessarily be used as part of this role. This allows dependencies between roles to be more clearly established.

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Deliver Goods’</td>
<td>Name of the role.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Deliver goods to customers …’</td>
<td>A brief description of the role.</td>
</tr>
<tr>
<td>Provided Services</td>
<td>Service(A8)[]</td>
<td>{'DeliveryRequest’, ‘DeliverGoods’}</td>
<td>Services provided by the role.</td>
</tr>
<tr>
<td>Used Services</td>
<td>Service(A8)[]</td>
<td>{'DispatchParcel’}</td>
<td>Services necessary for the role.</td>
</tr>
</tbody>
</table>

Table 4.9 – Role (A5)

4.7.3.6 Sensor (A6)

This concept models the means by which an agent senses environmental information. In NUMAP, environmental information is broadly defined, and includes user input, data accessed from files or databases, or any other data that is stored external to the agents in the system. Sensors may therefore have a variety of forms, from user interface components to mechanical sensors.

The sensor concept has the following properties (listed in Table 4.10):

Name: This is a brief, descriptive name used to identify the sensor.

Description: A brief description of the sensor, indicating its importance within the system.

Environment Element: The type of environmental element read by this sensor.
<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Account Database Reader’</td>
<td>Name of the sensor.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Read data from the Account Database …’</td>
<td>A brief description of the sensor.</td>
</tr>
<tr>
<td>Environment Element</td>
<td>Environment Element</td>
<td>{Account Record}</td>
<td>Type of data read in, specified as an environment element.</td>
</tr>
</tbody>
</table>

Table 4.10 – Sensor (A6)

4.7.3.7 **Effector (A7)**

This concept specifies the means by which an agent changes its environment. Effectors may have a variety of forms, from file writers to mechanical devices.

The effector concept has the following properties (listed in Table 4.11):

*Name:* This is a brief, descriptive name used to identify the effector.

*Description:* A brief description of the effector, indicating its importance within the system.

*Environment Element:* The type of environmental element affected by this effector.

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Account Database Writer’</td>
<td>Name of the effector.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Write data to the Account Database …’</td>
<td>A brief description of the effector.</td>
</tr>
<tr>
<td>Environment Element</td>
<td>Environment Element</td>
<td>{Account Record}</td>
<td>Type of environmental element that is changed</td>
</tr>
</tbody>
</table>

Table 4.11 – Sensor (A7)
4.7.3.8  Service (A8)

A service is a task that is performed by one agent on behalf of another agent. Services may be broadly defined, so in NUMAP’s analysis phase, they are simply defined by a name and description. The exact mechanism used to provide this service will be defined in subsequent phases.

The sensor concept has the following properties (listed in Table 4.12):

Name: This is a brief, descriptive name used to identify the service.

Description: A brief description of the service, indicating its importance within the system.

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Process payment’</td>
<td>Name of the service.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Processes payments on behalf of another agent …’</td>
<td>A description of the service.</td>
</tr>
</tbody>
</table>

Table 4.12 – Service (A8)

4.7.3.9  Plan (A9)

This represents a series of actions that will be used to achieve a goal or react to an event. This is analogous to actions in GBRAM, but the terminology used here is more consistent with that generally used in PRS-style agent systems. At this stage, the sensors, effectors and services to be used by the plan, and the goals it satisfies, are defined.

Plans have the following properties (shown in Table 4.13):

Name: This is the name used to identify the plan. It should be a brief, descriptive name denoting what task the plan performs.

Description: A brief description of the plan, indicating its importance within the system.
### Table 4.13 – Plan (A9)

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>Create account’</td>
<td>Name of the plan.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Create account …’</td>
<td>A description of the plan.</td>
</tr>
<tr>
<td>Sensors Used</td>
<td>Sensor(A6)[]</td>
<td>{DatabaseReader}</td>
<td>Sensors used by the plan.</td>
</tr>
<tr>
<td>Effectors Used</td>
<td>Effector(A7)[]</td>
<td>{DatabaseWriter}</td>
<td>Effectors used by the plan.</td>
</tr>
<tr>
<td>Services Used</td>
<td>Service(A8)[]</td>
<td>{DatabaseLookup}</td>
<td>Services used by the plan.</td>
</tr>
<tr>
<td>Satisfied Goals</td>
<td>Goal(A3)[]</td>
<td>{Account Created}</td>
<td>The goals satisfied by the plan.</td>
</tr>
</tbody>
</table>

*Sensors Used:* Indicates what data is required to perform the functionality of the plan, or more specifically, what sensors will be used to retrieve that data. A plan may require the use of more than one sensor.

*Effectors Used:* Indicates how the environment will be affected by the agent, or more specifically, which techniques will be used for interactions with the environment. A plan may require the use of more than one effector.

*Services Used:* Indicates which services will be required for use by the plan. A plan may require the use of more than one service.

*Satisfied Goals:* Indicates which goals will be satisfied by the plan. Unlike in GBRAM, a single plan may be effective for satisfying number of goals.

#### 4.7.3.10 Mapping the Requirements Phase to the Analysis Phase

The mappings between the Analysis Phase and the two Design Phases are presented in Appendix B.

As the Analysis phase includes concepts relevant to both inter-agent functionality and internal agent functionality, concepts from this phase are directly mapped to elements of both phases.
4.7.3.11 Summary

The above descriptions detail the information gathered during the analysis phase of the process. At the conclusion of the phase, a full agent-oriented analysis of the system will have been defined.
4.8 Organisational Design Phase

4.8.1 Purpose of the Organisational Design Phase

The organisational design phase of NUMAP involves defining the system at an inter-agent level. The interactions between agent types are defined, along with the services provided by each agent type.

The internal functionality for each agent is not defined at this stage, as internal design is separated into the distinct “Agent Design” phase. This separation is an important aspect of NUMAP’s modularity. The separation of the design of the agent interactions from the internal agent details allows for different agent design approaches to be used in place of BDI agents if required. It also allows for the creation of heterogeneous multiagent systems, where different types of agent, and even non-agent components, are integrated within the one system.

During this stage, concepts that were defined in the analysis phase are refined, and several additional concepts are added, to fully specify inter-agent behavior. Specifying interaction protocols for use by agents, and detailing the mechanisms for communication between agents are central to this.

At the conclusion of this phase, the developer will have an overall view of how agents are organised, and how they interact and collaborate to satisfy the requirements of the system.

At this phase of the process, the design is not tied to any particular agent implementation approach. This potentially allows for the design of heterogeneous agent systems, with different design modules being used for the internal specification of individual agents.

4.8.2 Concepts Defined in the Organisational Design Phase

At the conclusion of the analysis phase, a number of concepts were identified that pertain to the organisational design of the system. By this stage, the agent
types to be used in the system will have been identified, along with the goals they pursue, the roles they play, and the services they provide.

As development moves into the organisational design phase, these concepts may need to be refined, as the developer takes a more design-oriented view of the system. Some agent types may encapsulate too much functionality, and may need to be decomposed into smaller, more manageable entities, to assist with their lower-level design.

In refining the design at this stage, it is important to ensure the agent types retain their coherence, and each perform a clearly defined role within the system. This refinement will also apply to other concepts, such as roles and services. At this stage of the process, they should be defined in sufficient detail to represent the actual services provided by agents in the final system.

In addition to the refinement of the concepts introduced in the analysis phase, additional concepts are added to handle the details of communication between agent types.

4.8.3 Defining the Communication Concepts for Organisational Design

The developer must now fully specify how communications between agents will occur. Early in this stage, the services provided by agents will have been identified, but a specification detailing the interactions involved in delivering these services needs to be detailed.

In the Prometheus methodology, interaction diagrams are specified for each interaction scenario, and refined into interaction protocols, described using a modified Agent UML Sequence Diagram [72]. The actual information exchanged during communications is defined in terms of messages. The Tropos methodology takes a similar approach, using Agent UML Sequence Diagrams to represent the interaction protocols for agents.
This approach is a sensible one, as FIPA have defined a suite of interaction protocols [77, 131, 133, 135, 141] that cover a wide range of typical agent interactions. These can be used as a basis for interactions, serving as an effective library of patterns for agent communications. As such, NUMAP defines agent communications in terms of Interaction Protocols, and encourages the use of FIPA’s suite of interaction protocols where practical.

The actual format of the messages exchanged as part of these interaction protocols also needs to be defined. Messages in FIPA communication specifications are defined in terms of a “performativ e” for each communicative act; these performatives designate the type of communication that is occurring [131]. In NUMAP, the messages are defined in terms of such a performativ e, along with the actual content of the message defined in terms of parameters.

These two concepts, interaction protocols and messages are the basis for specifying agent communications in NUMAP. The environment elements that were identified during analysis will also be refined at this stage, serving as a basic ontology for communications.

4.8.4 Specification of Organisational Design Phase Concepts

As noted above, the concepts used in the organisational design phase primarily consist of refinements to the information identified during analysis. The goal of this phase is to add detail to these concepts to a degree where they fully define the interactions between agents within the system. This requires new concepts to specify communication details.
The following concepts will be defined for the organisational design phase:

- *Agent Types*
- *Messages*
- *Roles*
- *Services*
- *High-Level Goals*
- *Interaction Protocols*
- *Environment Elements*
- *Organisations*

These concepts will fully describe the interactions between agents in the system.

### 4.8.4.1 *Agent Type (O1)*

The agent type concept represents the kinds of agents in the system at an organisational level. At this level, a final decomposition of agents for the system should be established, reflecting the final list of agents that will be used in implementation. Stakeholders need not be defined, but should be taken into account when creating the final agent list. The focus in this phase is on defining each agent in terms of its interactions. Internal functionality will be very briefly described, but detail is left to subsequent phases.

The agent type concept has the following properties (shown in Table 4.14):

*Name:* This is the name used to identify the agent type. It should be a brief, descriptive name denoting the title of the agent.

*Description:* A brief description of the agent, indicating its importance within the system.

*Roles:* The roles played by the agent. Indicates what responsibilities the agent agrees to take, and therefore which services it will provide. At the
organisational design phase, the concept of roles and services is central, so these should be carefully considered.

Goals: The goals that the agent will pursue. At this stage, goals are only described with a simple overview, as mentioned below. A collection of Organisational Design Goals will be specified.

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Customer’</td>
<td>Name of agent.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Buys goods …’</td>
<td>A brief description of the agent</td>
</tr>
<tr>
<td>Roles</td>
<td>Role(O3)[]</td>
<td>{‘Manage customers’, ‘Order products’}</td>
<td>The roles played by the agent, i.e. its core responsibilities to other agents.</td>
</tr>
<tr>
<td>Goals</td>
<td>Goal(O5)[]</td>
<td>{‘User added’, ‘Stock replenished’}</td>
<td>The goals the agent is to achieve.</td>
</tr>
</tbody>
</table>

Table 4.14 – Agent (O1)

4.8.4.2 Message (O2)

This concept represents inter-agent messages at an organisational level. The message structure for communication is defined, in terms of the FIPA performatives used and the parameters contained within the message. This concept has the following properties (shown in Table 4.15):

Name: This is the name used to identify the message. It should be a brief, descriptive name denoting the type of message that is being defined.

Description: A brief description of the message, indicating its importance within the system.

Performative: This is a FIPA message performative [142] describing the type of communication that is occurring. It is represented by the name of the performative.
**Parameters:** This describes the parameters that the message may have. Data types for each parameter should be specified as part of the description of the parameter. The data type may be defined in terms of an EnvironmentElement.

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Status Update’</td>
<td>Name of the message type</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Provides an update of an agent’s status …’</td>
<td>A brief description of the message type</td>
</tr>
<tr>
<td>Performative</td>
<td>String</td>
<td>‘INFORM’</td>
<td>The FIPA performative for the message.</td>
</tr>
<tr>
<td>Parameters</td>
<td>String or</td>
<td>{'Status: String'}</td>
<td>The parameters carried by the message.</td>
</tr>
<tr>
<td></td>
<td>Environment-Element(O7)]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Table 4.15 – Message (O2)*

### 4.8.4.3  **Role (O3)**

This concept models roles that may be played by agents. Roles are defined in the same manner as in the analysis phase, but at this stage, they need to be considered from an implementation perspective. Roles should be well-defined, as they are crucial to describing the responsibilities of each agent.

The role concept has the following properties (shown in *Table 4.16*):

**Name:** This is a brief, descriptive name used to identify the role.

**Description:** A brief description of the role, indicating its importance within the system.

**Provided Services:** These are the services that an agent adopting this role commits to providing. More detail on services is provided below.

**Used Services:** These are the services that will necessarily be used as part of this role. This allows dependencies between roles to be more clearly established.
<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Deliver Goods’</td>
<td>Name of the role.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Deliver goods to customers …’</td>
<td>A brief description of the role.</td>
</tr>
<tr>
<td>Provided Services</td>
<td>Service(O4)[]</td>
<td>{'DeliveryRequest’, ‘DeliverGoods’}</td>
<td>Services provided by the role.</td>
</tr>
<tr>
<td>Used Services</td>
<td>Service(O4)[]</td>
<td>{'DispatchParcel'}</td>
<td>Services necessary for the role.</td>
</tr>
</tbody>
</table>

Table 4.16 – Role (O3)

4.8.4.4 Service (O4)

A service is some task that is performed by one agent on behalf of another agent. In NUMAP’s organizational design phase, services are defined in more detail than in the analysis phase. Services are one of the fundamental building blocks of agent communication, so careful attention needs to be taken in properly constructing the service framework for the system.

Services are defined with the following properties (defined in Table 4.17):

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Process payment’</td>
<td>Name of the service.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Processes payments on behalf of another agent …’</td>
<td>A description of the service.</td>
</tr>
<tr>
<td>Interaction Protocols</td>
<td>Interaction Protocol(O6)[]</td>
<td>{'REQUEST-RESPONSE’}</td>
<td>The interaction protocol(s) used when requesting the service.</td>
</tr>
<tr>
<td>ACL</td>
<td>String</td>
<td>‘FIPA-ACL’</td>
<td>The communication language to be used.</td>
</tr>
<tr>
<td>Properties</td>
<td>String[]</td>
<td>{'data: Payment’}</td>
<td>The properties to be supplied when requesting the service.</td>
</tr>
</tbody>
</table>

Table 4.17 – Service (O4)
Name: This is a brief, descriptive name used to identify the service.

Description: A brief description of the service, indicating its importance within the system.

Interaction Protocols: The FIPA interaction protocols used within the service. These are further described below.

ACL: The communication language to be used with the service. The communication language will be defined in a separate specification (e.g. for FIPA-ACL [142]), so this will simply store the name of the interaction protocol to be used.

Properties: The properties to be supplied when requesting the service. Data types for each property should be specified as part of the description of the property.

4.8.4.5 Goal (O5)

The goal concept represents an aim to be achieved by an agent or agents within the system. At the organisational design level, the internal functionality of the agent is not defined, but it is still useful to provide a list of the goals to be achieved to aid in understanding the purpose of the agent. At this stage, specifics are omitted, and a simple description of the goal is all that is provided.

The goal concept has the following properties (shown in Table 4.18):

Name: This is used to identify the goal. It should be a brief, descriptive name denoting what the goal is trying to achieve.

Type: This property may have one of two values. “Achievement” indicates that the goal is an achievement goal, “Maintenance” indicates that the goal is a maintenance goal.
### Key

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Account created’</td>
<td>Name of the goal.</td>
</tr>
<tr>
<td>Type</td>
<td>Achievement</td>
<td>‘Agent’</td>
<td>The goal type.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Create account …’</td>
<td>A brief description of the goal.</td>
</tr>
<tr>
<td>Parent</td>
<td>Goal(O5)</td>
<td>Handle Accounts</td>
<td>The parent goal in the goal hierarchy.</td>
</tr>
</tbody>
</table>

*Table 4.18 – Goal (O5)*

**Description:** A brief description of the goal, indicating its importance within the system.

**Parent:** Goals may be organised into a hierarchy. The parent goal of an agent must therefore be specified as part of the goal.

#### 4.8.4.6 Interaction Protocol (O6)

Interaction protocols show the sequence of interactions between agents. A number of existing interaction protocols are described in detail in FIPA documents [77, 131, 133, 135, 141]. In the case that an existing FIPA Interaction Protocol exists to match the desired agent interactions, this may simply be referenced.

However, it is likely there will be some interactions in the system that are complex, and cannot be adequately described using existing interaction protocols. In these situations, a new interaction diagram should be created, and referenced by the concept. This should be specified using an Agent UML Sequence Diagram [72].

The Interaction Protocol concept has the following properties (shown in Table 4.19):
<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘REQUEST’</td>
<td>Name of the interaction protocol.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Requests some feedback from another agent …’</td>
<td>A brief description of the interaction protocol.</td>
</tr>
</tbody>
</table>

**Table 4.19 - Interaction Protocol (O6)**

*Name:* This is a brief, descriptive name used to identify the organisation.

*Description:* A brief description of the organisation, indicating its importance within the system.

*Document:* A reference to an external document defining the interaction protocol.

**4.8.4.7 Environment Element (O7)**

The Environment element concept models the relevant types of data that exist in the environment of the system. This is defined at the organisational design phase, because it may assist when defining messages and services – the collection of environment elements can serve as a simple ontology for communicating information about the environment.

Environment elements will be largely imported from the preceding phase, but refinement will likely occur during organisational design.
This concept has the following properties (shown in Table 4.20):

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Accounts Database Record’</td>
<td>Name of the element.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Stores account information …’</td>
<td>A brief description of the element.</td>
</tr>
<tr>
<td>Attributes</td>
<td>String[]</td>
<td>{‘Name’, ‘Balance’}</td>
<td>Attributes of the element.</td>
</tr>
<tr>
<td>Parent</td>
<td>Environment Element(O7)</td>
<td>Database System</td>
<td>The parent element in the hierarchy.</td>
</tr>
</tbody>
</table>

Table 4.20 – Environment Element (O7)

*Name*: This is the name used to identify the environment element. It should be a brief, descriptive name denoting what the element is, or what data it contains.

*Description*: A brief description of the environment element, indicating its importance within the system.

*Attributes*: A list of the attributes of the environment element, listing the properties that are contained within the element. These are defined in a similar manner to specifying the attributes of a business object in object-oriented design, and are provided in text format.

*Parent*: Environment elements may be nested into a hierarchy. This property indicates the parent (more general) element of the one that is being specified.

### 4.8.4.8 *Organisation* (O8)

This concept models logical groupings of agents. Organisations are used to componentise the system, grouping tightly-coupled agents, and allowing a higher-level view of the system to be presented. The organisations at this phase will be based upon those identified during analysis, but may change as implementation considerations are taken into account.
This concept has the following properties (shown in Table 4.21):

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Commerce Agents’</td>
<td>Name of the organisation.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Agents buying/selling goods …’</td>
<td>A brief description of the organisation.</td>
</tr>
<tr>
<td>Interaction Organisations</td>
<td>Organisation(O8)[]</td>
<td>{Courier Agents}</td>
<td>The organisations this one interacts with.</td>
</tr>
<tr>
<td>Member Agents</td>
<td>Agent(O1)</td>
<td>{Supplier, Retailer, Customer}</td>
<td>The agent types that are a member of the organisation.</td>
</tr>
<tr>
<td>Parent</td>
<td>Organisation(O8)</td>
<td>{Simulation Agents}</td>
<td>The parent organisation.</td>
</tr>
</tbody>
</table>

**Table 4.21 - Organisation (O8)**

*Name:* This is a brief, descriptive name used to identify the organisation.

*Description:* A brief description of the organisation, indicating its importance within the system.

*Interaction Organisations:* This is a list of organisations with which this one interacts, that is, the organisations with which this organisation has a dependency. These interactions will generally be through communications with, or the use of a service from, an agent in the other organisation.

*Member Agents:* A list of the agents that are a member of the organisation. An agent may be a member of multiple organisations.

*Parent:* Organisations may be organised into a hierarchy. The parent organisation must therefore be specified if it exists.
4.8.4.9  **Mapping the Organisational Design Phase to the Detailed Design Phase**

As the majority of concepts will have already been mapped directly from the analysis phase to the detailed design phase, there is little mapping to be done between these two phases. However, any new agent types introduced during organisational design will need to be added to the detailed design phase, and mappings updated to reflect the relationship between the agent types in these phases.

The details of how this is achieved are presented in Appendix B.

4.8.4.10  **Summary**

During the organisational design phase of the process, the inter-agent functionality of the system will have been fully specified. At the conclusion of this phase, development is ready to proceed to detailed agent design.
4.9 Agent Design Phase

4.9.1 Purpose of the Agent Design Phase

The Agent Design phase is a modular phase of NUMAP, designed to be independent from the rest of the process. As there are a wide variety of techniques available for designing agents, a variety of modules could be created for this phase of the process.

Because the initial goal of the NUMAP process was the creation of a design process for BDI multiagent systems, a module has been created that supports the definition of BDI agents. This module closely maps to the concepts used in implementing agents in modern BDI implementation platforms, such as JACK [65] and Jadex[39].

The purpose of this phase is to specify agents in sufficient detail that they can be readily implemented. There should be a very close correlation between the design at the conclusion of this phase, and the actual implementation of the agents.

4.9.2 Concepts Defined in the Agent Design Phase

The agent design phase builds upon the agent types identified during the analysis and organisational design phases, specifying the low-level detail of how they are to be realised.

The concepts of sensors, effectors, plans and goals are also carried-over from analysis. These will be further refined during this phase.

Additionally, during this phase, the agent’s beliefs are defined, its mechanisms for selecting plans are specified, and “agent capabilities” are defined, to aid in code reuse.
4.9.2.1 Defining Goals, Plans, Reactions at an Agent Level

At the heart of any BDI multiagent system is an agent’s desires and intentions. These are typically referred to as goals and plans. The goals of an agent specify its purpose – what it is to achieve. The plans specify the actions it should take to go about satisfying these goals. Together, these two concepts specify the “proactive” portion of the agent’s behaviour, i.e. their goal-directed behaviour.

However, agents do not simply act proactively. They will also have a reactive component [38]. This allows them to react in a timely fashion to events that occur either internally or externally. These events may be an error condition (e.g. failure of a goal), or an external stimulus detected by an agent sensor.

At the agent design phase, therefore, we extend the goals and plans identified at earlier stages of the process, and also add the concept of reactions.

4.9.2.2 Defining Selection Rules at an Agent Level

It is not uncommon within BDI agents to have multiple plans that can satisfy a goal. It is therefore necessary to have a formal mechanism for specifying which plan will be selected in such a situation. In fact, in the context of the Jadex platform, it has been noted that the “core of a BDI architecture is obviously the mechanism for plan selection” [143].

This aspect of agent design is often overlooked in multiagent systems processes. One of the few processes to consider this aspect is Prometheus. It introduces the notion of “overlap” – the situation in which multiple plans overlap in satisfying a particular trigger [22]. However, this concept, while mentioned, is not explicitly modelled.

A similar issue arises when an agent is required to use a service offered by another agent. There may be multiple providers available for the particular
service, so a decision needs to be made on which to utilise. In NUMAP, this is referred to as “delegation selection”.

Therefore, we introduce the additional concepts of plan selection and delegation selection to the NUMAP process.

4.9.2.3 Defining Beliefs At An Agent Level

Beliefs are a fundamental part of any BDI agent. They represent the agent's view of its environment, of itself, and of other agents. Beliefs in a multiagent system are not absolute – they simply represent what an agent holds to be true, given the information it has available to it [144]. Beliefs can, and should, be revised by agents given new information, whether that information comes from their sensors, from other agents, or through an update to a belief they hold internally.

The nature of beliefs, and how they should be updated, therefore depends on the kind of information being represented. NUMAP defines three types of belief: “Introspective” beliefs are beliefs an agent holds about some internal property it possesses. “Social” beliefs are regard properties of other agents, and “sensed” beliefs relate to information detected from the environment, through sensors.

In NUMAP, these are represented using the belief concept.

4.9.2.4 Defining Capabilities at an Agent Level

Capabilities are a mechanism for clustering agent functionalities for the purposes of reuse and to avoid duplication between agents. They were first introduced by Busetta et al [145], and are a core part of agent platforms such as JACK [65] and Jadex[39].

In NUMAP, the concept of a capability encapsulates a cluster of beliefs, goals, reactions and plans.
4.9.3  **Specification of Agent Design Phase Concepts**

Using the above concepts, we can fully describe a BDI agent in a manner that can be directly implemented on a BDI platform. The concepts from the analysis phase have been augmented with new concepts to track an agent's beliefs, its reactions, and how it selects alternatives.

The following concepts will be defined for the agent design phase:

- **Agent Types**
- **Plans**
- **Goals**
- **Reactions**
- **Plan Selection Rules**
- **Delegation Selection Rules**
- **Sensors**
- **Effectors**
- **Beliefs**
- **Capabilities**

These concepts are specified as follows.

4.9.3.1  **Agent Type (D1)**

This concept defines the internal aspects of the agent, and is needed to define how the agent type will function internally.

The agent type concept has the following properties (defined in Table 4.22)

*Name*: This is the name used to identify the agent. It should be a brief, descriptive name denoting the title of the entity or the role they play within the system.
**Description:** A brief description of the agent, indicating its importance within the system.

**Beliefs:** The beliefs held by the agent.

**Goals:** The goals that the agent may pursue. The concept of goals in this phase is further defined below.

**Reactions:** A collection of reactive behaviours for the agent. The concept of reactions is further defined below.

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Customer’</td>
<td>Name of agent or stakeholder</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Buys goods …’</td>
<td>A brief description of the agent or stakeholder</td>
</tr>
<tr>
<td>Beliefs</td>
<td>Belief(D9)[ ]</td>
<td>{'Current Temperature’, ‘Stock Level’}</td>
<td>The beliefs held by the agent.</td>
</tr>
<tr>
<td>Goals</td>
<td>Goal(D3)[ ]</td>
<td>{'User added’, ‘Stock replenished’}</td>
<td>The goals the agent may achieve.</td>
</tr>
<tr>
<td>Reactions</td>
<td>Reaction(D4)[ ]</td>
<td>{'Handle message’}</td>
<td>The reactions the agent may perform.</td>
</tr>
<tr>
<td>Plans</td>
<td>Plan(D2)[ ]</td>
<td>{'WriteFile’, ‘ReadData’}</td>
<td>Plans used by an agent to achieve its goals.</td>
</tr>
<tr>
<td>Capabilities</td>
<td>Capability(D10)[ ]</td>
<td>{'Order management’}</td>
<td>The capabilities used by the agent.</td>
</tr>
<tr>
<td>Sensors</td>
<td>Sensor(D7)[ ]</td>
<td>{'Database Connector’, ‘User dialog’}</td>
<td>Sensors used by the agent to gather information from its environment.</td>
</tr>
<tr>
<td>Effectors</td>
<td>Effector(D8)[ ]</td>
<td>{'File writer’, ‘Motor controller’}</td>
<td>Effectors used by the agent to alter the state of its environment.</td>
</tr>
</tbody>
</table>

| Table 4.22 - Agent (D1) |

**Plans:** The plan library for the agent. This is the collection of plans that are used by the agent to achieve its goals. The concept of plans is further defined below.

**Capabilities:** The capabilities used by an agent. Capabilities are a collection of agent properties that may be inherited, and used by, a number of agents. The concept of capabilities is further defined below.
**Sensors:** A collection of “sensors” that are used by the agent to obtain information from its environment. The concept of sensors is further defined below.

**Effectors:** A collection of “effectors” that are used by the agent to alter its environment. The concept of effectors is further defined below.

### 4.9.3.2 Plan (D2)

This represents a plan that will be used to achieve a goal or react to an event.

Plans in the detailed design phase have the following properties (shown in Table 4.23):

**Name:** This is used to identify the plan. It should be a brief, descriptive name denoting what task the plan performs.

**Description:** A brief description of the plan, indicating its importance within the system.

**Sensors Used:** Indicates what data is required to perform the functionality of the plan, or more specifically, which sensors will be used to retrieve that data. A plan may require the use of more than one sensor.

**Effectors Used:** Indicates how the environment will be affected by the effectors, or more specifically, which effectors will be used for this purpose. A plan may require the use of more than one effector.

**Services Used:** Indicates what services will be required to be used by the plan. A plan may require the use of more than one service.

**Satisfied Goals:** Indicates which goals will be satisfied by the plan. Unlike in GBRAM, a single plan may be effective for satisfying number of goals.
Satisfied Reactions: Indicates which goals will be satisfied by the plan. Unlike in GBRAM, a single plan may be effective for satisfying a number of goals.

Parameters: Indicates the parameters that may be supplied to, and used by, the plan.

Procedure: A full outline of the procedures to be followed in implementing the plan. This may take the form of an ordered list of steps. Exceptions (e.g. failure) should be mentioned.

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>Create account’</td>
<td>Name of the plan.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Create account …’</td>
<td>A description of the plan.</td>
</tr>
<tr>
<td>Sensors Used</td>
<td>Sensor(D7)[]</td>
<td>{DatabaseReader}</td>
<td>Sensors used by the plan.</td>
</tr>
<tr>
<td>Effectors Used</td>
<td>Effector(D8)[]</td>
<td>{DatabaseWriter}</td>
<td>Effectors used by the plan.</td>
</tr>
<tr>
<td>Services Used</td>
<td>Service(H4)[]</td>
<td>{DatabaseLookup}</td>
<td>Services used by the plan.</td>
</tr>
<tr>
<td>Satisfied Goals</td>
<td>Goal(D3)[]</td>
<td>{Account Created}</td>
<td>The goals satisfied by the plan.</td>
</tr>
<tr>
<td>Satisfied Reactions</td>
<td>Reaction(D4)[]</td>
<td>{Account Event}</td>
<td>The reactions satisfied by the plan.</td>
</tr>
<tr>
<td>Parameters</td>
<td>String[]</td>
<td>‘Account Name: String’</td>
<td>Parameters used by the plan.</td>
</tr>
<tr>
<td>Procedure</td>
<td>String</td>
<td>‘1: Step 1, 2: Step2’</td>
<td>Procedure to be followed by plan.</td>
</tr>
</tbody>
</table>

Table 4.23 – Plan (D2)

4.9.3.3 Goal (D3)

This concept represents the goals to be achieved within the system. It defines the proactive aspect of an agent.

The goal concept has the following properties (shown in Table 4.24):

Name: This is the name used to identify the goal. It should be a brief, descriptive name denoting what the goal is trying to achieve.
**Type**: This property may have one of two values. "Achievement" indicates that the goal is an achievement goal, "Maintenance" indicates that the goal is a maintenance goal.

**Description**: A brief description of the goal, indicating its importance within the system.

**Failure Conditions**: These represent specific situations under which a goal may fail, written as a textual description. A goal may have several failure conditions.

**Preconditions**: As with the requirements phase, these are a set of conditions that must be met before a goal can start being pursued. These are to be written as a textual description. A goal may have several preconditions.

**Postconditions**: As with the requirements phase, these are the conditions that must be met for a goal to be considered to be achieved. These are to be written as a textual description. A goal may have several postconditions.

**Parent**: Goals may be organised into a hierarchy. The parent goal of an agent must therefore be specified as part of the goal.

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>'Account created'</td>
<td>Name of the goal.</td>
</tr>
<tr>
<td>Type</td>
<td>Achievement</td>
<td>'Agent'</td>
<td>The goal type.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>'Create account …'</td>
<td>A brief description of the goal.</td>
</tr>
<tr>
<td>Failure Condition</td>
<td>String[]</td>
<td>{'Username exists'}</td>
<td>Specific situations that will cause a goal to fail.</td>
</tr>
<tr>
<td>Preconditions</td>
<td>String[]</td>
<td>{'System online'}</td>
<td>Situations that must have arisen before the goal is attempted to be achieved.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>String[]</td>
<td>{'Account processed'}</td>
<td>Specific conditions that must be met for the goal to be achieved.</td>
</tr>
<tr>
<td>Parent</td>
<td>Goal(D3)</td>
<td>Handle Accounts</td>
<td>The parent goal in the goal hierarchy.</td>
</tr>
</tbody>
</table>

*Table 4.24 – Goal (D3)*

150
This concept represents reactive behaviour. It defines situations where an immediate response (via a plan) may be required, rather than deliberative reasoning.

The reaction concept has the following properties (shown in Table 4.25):

**Name:** This is the name used to identify the reaction. It should be a brief, descriptive name denoting the situation in which the reaction occurs.

**Type:** This property may have one of two values. “Message” indicates that the reaction is in response to a message from another agent, “Event” indicates that it is in response to some other event.

**Description:** A brief description of the goal, indicating its importance within the system.

**Events:** A brief description of the event that triggers the reaction. More than one event trigger may be listed.

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Database Failure’</td>
<td>Name of the reaction.</td>
</tr>
<tr>
<td>Type</td>
<td>Message</td>
<td>Event</td>
<td>The reaction type.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘DB error …’</td>
<td>A brief description of the reaction.</td>
</tr>
<tr>
<td>Events</td>
<td>String[ ]</td>
<td>{‘No Access to DB’, ‘Database Exception’}</td>
<td>The event(s) that trigger the reaction.</td>
</tr>
</tbody>
</table>

Table 4.25 - Reaction (D4)
4.9.3.5  **Plan Selection Rule (D5)**

This concept describes a rule that is used to select which plan is to be chosen to fulfil a particular goal or reaction.

The plan selection rule concept has the following properties (shown in *Table 4.26*):

**Name:** This is the name used to identify the selection rule.

**Description:** A brief description of the selection rule.

**Rule Definition:** Describes the rule that is to be implemented. This may be as a plain text description, using pseudocode, or any other definition the user prefers.

**Applicable Goals:** The goals for which this selection rule is relevant.

**Applicable Reactions:** The reactions for which this selection rule is relevant.

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Select Delivery Plan’</td>
<td>Name of the selection rule.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Select the plan to be used for delivery…’</td>
<td>A brief description of the rule.</td>
</tr>
<tr>
<td>Rule Definition</td>
<td>String</td>
<td>‘Choose any plan that has not experienced failure in this agent…’</td>
<td>Defines the rule to be implemented.</td>
</tr>
<tr>
<td>Applicable Goals</td>
<td>Goal(D3)[]</td>
<td>{'Deliver'}</td>
<td>The goals that the selection rule applies to.</td>
</tr>
<tr>
<td>Applicable Reactions</td>
<td>Reaction(D4)[]</td>
<td>{'DeliverEvent'}</td>
<td>The reactions that the selection rule applies to.</td>
</tr>
</tbody>
</table>

*Table 4.26 – Plan Selection Rule (D5)*
4.9.3.6 Delegation Selection Rule (D6)

This concept describes a rule that is used to select which agent, providing a service, is to be chosen when that service is required.

The delegation selection rule concept has the following properties (shown in Table 4.27):

Name: This is the name used to identify the selection rule.

Description: A brief description of the selection rule.

Rule Definition: Describes the rule that is to be implemented. This may be as a plain text description, using pseudocode, or any other definition the user prefers.

Applicable Services: The services for which this selection rule is relevant.

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Select Supplier’</td>
<td>Name of the selection rule.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Select the supplier to purchase from…’</td>
<td>A brief description of the rule.</td>
</tr>
<tr>
<td>Rule Definition</td>
<td>String</td>
<td>‘Choose the cheapest supplier for the order…’</td>
<td>Defines the rule to be implemented.</td>
</tr>
<tr>
<td>Applicable Services</td>
<td>Service(H4)[]</td>
<td>{'SupplierOrder'}</td>
<td>The services that the selection rule applies to.</td>
</tr>
</tbody>
</table>

Table 4.27 – Delegation Selection Rule (D6)

4.9.3.7 Sensor (D7)

This concept models the methods by which an agent senses environmental information. These may be based on the sensors identified during analysis, but should be refined as required for the needs of the agent implementation.
The sensor concept has the following properties (shown in Table 4.28):

_Name:_ This is a brief, descriptive name used to identify the sensor.

_Description:_ A brief description of the sensor, indicating its importance within the system.

_Environment Element:_ The kind of environmental element read by this sensor.

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Account Database Reader’</td>
<td>Name of the sensor.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Read data from the Account Database …’</td>
<td>A brief description of the sensor.</td>
</tr>
<tr>
<td>Environment Element</td>
<td>Environment Element (H7)[[]]</td>
<td>{Account Record}</td>
<td>Type of data read in, specified as an environment element.</td>
</tr>
</tbody>
</table>

Table 4.28 – Sensor (D7)

4.9.3.8 Effector (D8)

This concept models the means by which an agent changes its environment. These may be based on the effectors identified during analysis, but should be refined as required for the needs of the agent implementation.

The effector concept has the following properties (shown in Table 4.29):

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Account Writer’</td>
<td>Name of the effector.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Write data to the Account Database …’</td>
<td>A brief description of the effector.</td>
</tr>
<tr>
<td>Environment Element</td>
<td>Environment Element (H7)[[]]</td>
<td>{Account Record}</td>
<td>Type of environmental element that is changed</td>
</tr>
</tbody>
</table>

Table 4.29 – Effector (D8)
Name: This is a brief, descriptive name used to identify the effector.

Description: A brief description of the effector, indicating its importance within the system.

Environment Element: The type of environmental element affected by this effector.

4.9.3.9 Belief (D9)

This concept models a belief for an agent. This is a fundamental aspect of the agent; beliefs are used by plans (and possibly by selection rules) and are analogous to the attributes of an object oriented class.

The belief concept has the following properties (shown in Table 4.30):

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Stock Level’</td>
<td>Name of the belief.</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Stores the current stock level for a product …’</td>
<td>A brief description of the belief.</td>
</tr>
<tr>
<td>Data Type</td>
<td>String</td>
<td>‘Integer’</td>
<td>The data type of the belief.</td>
</tr>
<tr>
<td>Belief Type</td>
<td>Introspective</td>
<td>[Introspective]</td>
<td>Type of belief.</td>
</tr>
<tr>
<td></td>
<td>Social</td>
<td>Sensed</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.30 – Belief (D9)

Name: This is a brief, descriptive name used to identify the belief.

Description: A brief description of the belief, indicating its importance within the system.

Data Type: Indicates the data type of the belief.
Belief Type: This describes the kind of belief that is defined. “Introspective” beliefs describe a belief about an internal property of the agent, “Social” beliefs describe a belief about another agent, or the agent society in general, and “Sensed” beliefs describe a belief about the state of the agent’s environment.

The type of belief has a large influence on how the belief is updated. Introspective beliefs are updated internally, tracking the internal state of the agent. Social beliefs are updated in response to communications from another agent. Sensed beliefs are updated in response to a sensor input from the environment.

4.9.3.10 Capability (D10)

Capabilities are a collection of agent properties that may be inherited, and used by, a number of agents. In NUMAP, developers are encouraged to use capabilities to help in ensuring that a goal meets the responsibilities of its role. Services required for the role can be packaged into capabilities, which can then be inherited by multiple agents.

The capability concept has the following properties (shown in Table 4.31):

Name: This is the name used to identify the capability. It should be a brief, descriptive name denoting the purpose of the capability.

Description: A brief description of the capability, indicating its importance within the system.

Beliefs: The beliefs held by the capability.

Goals: The goals that the capability may pursue.

Reactions: A collection of reactive behaviours for the capability.
*Plans:* The plan library for the capability. This is the collection of plans that are used by the capability to achieve its goals.

<table>
<thead>
<tr>
<th>Key</th>
<th>Domain</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>‘Customer’</td>
<td>Name of agent or stakeholder</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>‘Buys goods ...’</td>
<td>A brief description of the agent or stakeholder</td>
</tr>
<tr>
<td>Beliefs</td>
<td>Belief(D9)[0]</td>
<td>{'Current Temperature',</td>
<td>The beliefs held by the agent.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Stock Level'}</td>
<td></td>
</tr>
<tr>
<td>Goals</td>
<td>Goal(D3)[0]</td>
<td>{'User added',</td>
<td>The goals the agent may achieve.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Stock replenished'}</td>
<td></td>
</tr>
<tr>
<td>Reactions</td>
<td>Reaction(D4)[0]</td>
<td>{'Handle message'}</td>
<td>The reactions the agent may perform.</td>
</tr>
<tr>
<td>Plans</td>
<td>Plan(D2)[0]</td>
<td>{'WriteFile',</td>
<td>Plans used by an agent to achieve its goals.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'ReadData'}</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.31 – Capability (D10)

4.9.3.11 Summary

During the agent design phase of the process, the internal reasoning and functionality of the agent has been specified. At the conclusion of this phase, implementation is ready to commence.

4.10 Implementation Phase

Implementation of the system is performed in the desired implementation environment. Assistance with this is offered via code generation in the support tool, and is further discussed in Chapter 5.
4.11 Dependency Tracking

The design of NUMAP allows for the tracking of dependencies in two distinct ways:

1. Tracking dependencies within a phase of the process.
2. Tracking dependencies between phases of the process.

The first form of dependency tracking highlights how a change to one entity (e.g. an agent or goal) will affect other entities within that phase, either directly or indirectly.

The second form of dependency tracking highlights how a change to an element within one phase will affect elements in subsequent phases.

4.11.1 Intra-Phase Dependency Tracking

Tracking dependencies within a phase is a relatively straightforward process to conceptualise. Changes to any modelled entity within that phase will have an impact on other entities that make a reference to it. For example, if an agent has a specific goal defined, making a change to that goal will have a direct impact on the agent in question. Therefore, that agent has a dependency on the goal in question.

Additionally, any entities that make reference to that agent would then be potentially affected by the goal change, due to their dependency on the agent for which the changed goal is defined. Therefore, the dependency relationships within phases take the form of a dependency tree.

4.11.1.1 Requirements Phase Dependencies

The potential dependency relationships between concepts in the requirements phase can be expressed simply, as there are only three concepts defined within the GBRAM requirements phase.
Changes to a goal may affect their related agents and actions within the GBRAM requirements phase.

The dependencies are shown in Fig. 4.2.

One fact that is of interest, as mentioned in Section 4.7.3.2 is that the relationship between agents and goals within the GBRAM requirements process is the inverse of what would be expected.

In the GBRAM approach, as originally defined by Anton, goals are defined as the primary concepts for requirements analysis. A number of agents and stakeholders may be associated with each goal, listed as properties of the goal. In NUMAP, to retain consistency with GBRAM, this approach is used for the requirements phase.

However, in designing NUMAP, it was found that the inverse way of linking goals to agents was far more intuitive. In BDI systems, each goal is pursued by a single agent acting autonomously. Goals are therefore a property that “belong” to an agent, so in subsequent phases, this relationship is inverted. This task is aided by the NUMAP support tool.
4.11.1.2 Analysis Phase Dependencies

In the analysis phase, the system is refined to a larger extent than it was during requirements, and a number of additional concepts are introduced. This leads to a more complex set of dependency relationships between concepts.

In this phase, the key element that may be affected by changes is the agent. Changes to sensors, effectors, plans, roles or goals may have a direct impact on an agent that makes reference to them. Changes to a service will affect any role that uses those services, and may in turn have a flow-on impact to agents, and the organisations that contain those agents.

The dependencies are shown in Fig. 4.3.
4.11.1.3 Organisational Design Phase Dependencies

Within the organisational design phase, the key focus is on communication between agents. The dependency relationships for the phase reflect this. Changes to a message, or the environment element used to describe the data within that message, will have a considerable flow-on effect to interaction protocols, services, roles and agents that rely on that message.

The dependencies are shown in Fig. 4.4.

![Figure 4.4 – Organisational Design Phase Dependencies](image)

4.11.1.4 Detailed Design Phase Dependencies

In the detailed design phase, agents form the central abstraction used for design. Therefore, they have a wide range of dependencies on the various concepts used during the phase.
Changes to a belief, sensor, plan or goal, for example, will have an obvious impact on any agent that utilises that element for its functionality.

The dependencies are shown in Fig. 4.5.

![Diagram of Agent Design Phase Dependencies](Image)

**Figure 4.5 – Agent Design Phase Dependencies**

### 4.11.1.5 Implementing Intra-Phase Dependencies

Intra-phase dependency tracking has been implemented in the NUMAP support tool. The support tool provides this functionality by taking as an input the entity (goal, agent, plan, etc) to be analysed. It then checks all modelled elements in the particular phase for any references to that particular entity. When a reference is discovered, that element is added to the dependency list, after which the checking continues recursively.
4.11.2 Inter-Phase Dependency Tracking

A more novel form of dependency tracking is offered for discovering dependency relationships between different phases of the process.

This mechanism takes advantage of the transitions that occur between phases in NUMAP, mapping concepts from one phase to those in the next. To effectively use this mechanism, the support tool should be used to assist in the transitions between phases. The “guides” offered in the NUMAP support tool allow concepts to be refined as they are mapped from one phase to the next. The options for refinement for each element of the process are:

- Add unchanged. The mapping is performed automatically, with the element being translated to the corresponding element in the next phase of the process.
- Split the entity. For example, a single goal in one phase may be refined into several goals in the next, while preserving the mapping relationship between the phases.
- Merge the entity. For example, two goals in one phase may be consolidated into a single goal in the next, while preserving the mapping relationship between the phases.

For example, when proceeding from the requirements phase to the analysis phase, several goals would likely have been defined. One of these goals may be called “Fulfil Order”. In refining the model for the analysis phase, this goal may be split into two separate goals, “Prepare Order” and “Ship Order”. The inter-phase dependency relationship between these goals will be preserved, with both of the goals in the analysis phase having an inter-phase dependency on “Fulfil Order”. 
Figure 4.6 – Inter-Phase and Intra-Phase dependencies

As an illustration, the diagram in Fig. 4.6 shows how dependencies may be mapped in both an intra-phase and inter-phase manner, using a simple example considering only the requirements and analysis phases. Inter-phase relationships are depicted with double-line arrows, intra-phase with single-line arrows.

The inter-phase dependency tracking mechanism provides a novel way to assess how changes will impact the system, taking advantage of NUMAP's unique method of handling transitions between phases.

4.12 Summary

NUMAP addresses the aims stated at the beginning of Chapter 4, in the following ways:

1. *End-to-end support for the software development lifecycle:* The process handles all components of design, from requirements through to implementation, as well as providing dependency tracking to assist with iterative development.
2. *Grounding in real-world agent design concepts:* NUMAP maps closely to concepts used in real-world agent implementation platforms, and provides a comprehensive agent design phase for BDI agents.

3. *Goal-based requirements analysis:* The default process for NUMAP makes use of GBRAM, but there is the flexibility to use other techniques if desired.

4. *Modularity and process tailoring:* The process has been designed in a modular fashion, so it is not tightly coupled to any particular Agent Design phase or Requirements Analysis technique.

5. *Practicality and detail:* NUMAP takes a detailed approach to specifying agent systems. All concepts have been specified in detail, and a support tool is offered to aid in practical use of the process.

6. *Inclusion of a comprehensive support tool:* A comprehensive support tool is provided, to be discussed in the following chapter.
Chapter 5  NUMAP Support Tool Implementation

5.1  Introduction

The experience of following a software engineering process is simpler and less error prone if a support tool exists to aid the user with the tasks in each phase of development. When designing NUMAP, such a support tool was seen as an integral part of the process.

The NUMAP support tool was designed in conjunction with the process itself, and provides a range of functionality specific to agent development to assist the software engineering practitioner with the task of designing and implementing a multi-agent system.

5.2  Overview of the NUMAP Support Tool

The GUI for the NUMAP support tool is divided into three main sections, depicted in Fig 5.1.

At the top of the screen is a file menu, which provides access to global operations such as opening files, saving files, versioning of files, dependency tracking and document generation. These operations can be performed at any time.

The left part of the screen shows the concept/phase tree. This displays a list of the phases in the process and concepts that are used within that phase. Along with nodes representing concepts within the system, this tree also shows nodes for performing functions within a phase. Functionality such as code generation and diagram access are accessed via this tree. A separate sub-tree for accessing phase transition guides is also available via this pane.
Adding an element, such as a goal, can be performed by selecting a concept from the list. This will present a blank form, in which data about the element can be entered.

To modify an existing element, it is selected from the list beside the form. Elements for concepts that use a hierarchy, such as goals, are presented in tree format, rather than a simple list.

**5.3 Features of the NUMAP Support Tool**

The NUMAP support tool supports a number of key features to ensure its effectiveness in guiding users. These features allow it to provide effective support to users as they proceed through the development process.

This section provides an overview of these features. Further details will be specified in the rest of this chapter.
5.3.1 Modularity

NUMAP is a modular process, so its support tool is also designed in a modular fashion. The tool has been designed to be extensible, in order to support any future modules that may be developed for NUMAP.

Phases in the process are separated into self-contained software modules. This feature is particularly important in light of NUMAP’s ability to support different requirements elicitation techniques and agent design approaches. New modules can be created independently, without requiring alteration to existing modules.

5.3.2 Efficient Data Entry

Data can be entered in a fast, accurate, and efficient manner, with all data for each process element specified in a single form within the user interface. Validation on each field, and use of dropdown boxes where appropriate, ensures the data that is entered is in the correct format.
Rapid data entry is vital, as the NUMAP process supports extremely detailed specification of the concepts used in its design. This complexity is required to provide effective coverage of BDI concepts, but also requires a large amount of data to be entered by users of the process. As such, one of the primary goals in developing the NUMAP support tool was to allow users to enter data in a fast and efficient manner.

### 5.3.3 Data Reuse Between Phases

The NUMAP support tool provides functionality to assist users with transitioning between phases of the process. Relevant data is automatically entered into subsequent phases of the process if the user chooses to do so during the phase transition process. This allows data entry to be streamlined, as users are not required to start each phase with a “blank slate”.

![Figure 5.3 – Entering details for a goal](image-url)
5.3.4 Visualisation

In order to better make use of the data that is specified for the multi-agent system, some form of visualisation is desirable [146]. NUMAP provides support for this. Diagrams form part of each phase module within the support tool.

---

**Figure 5.4 – Transitioning between phases**

---

**Figure 5.5 – Goal-agent diagram**

170
The NUMAP support tool can generate diagrams from the data that has been entered by the user. This support is extensible, allowing new diagrams to be created for existing NUMAP modules, and also allowing diagrams to be created for new phases that are added to the process.

Diagrams are created using JGraph [147]. A number of diagrams have been created within the support tool as a proof-of-concept for this functionality. For example, in the GBRAM-based requirements phase of the default NUMAP process, two of the diagrams provided are Goal-Agent Diagram and the Subgoal Diagram.

The Goal-Agent Diagram (Fig. 5.5) shows the relationships between goals and the agents that are associated with them. Goals are represented as plain rectangles, whereas agents have a circle-and-triangle icon to distinguish them.

The Subgoal Diagram (Fig 5.6) shows subgoal relationships. This diagram is in tree-form, with a “System Goal” as the root node.

![Subgoal diagram](image)

Figure 5.6 – Subgoal diagram
5.3.5 Code Template Generation

The NUMAP support tool provides a mechanism for flexible code generation. This generates code templates based on the data that has been entered during the agent design phase of the process. The code generation is able to support a number of agent runtime environments – initially JACK [125] and Jadex [39] have support (see Figures 5.8 and 5.9).

It is possible to add support for other agent environments, and to adapt the code generation depending on which agent design module is being used. The code generation scheme is extensible, with the ability to support different agent runtime environments, and even different agent design modules that may be developed.

When a user selects to generate code, they are presented with a dialog box, by which they can select the agent implementation environment for which they wish to generate code, then an output path for the code.

![Image](image.png)

Figure 5.7 – Code generation selection
Jadex Code Snippet:

```
<goals>
  <maintaingoalref name="df_keep_registered">
    <concrete ref="dfcap.df_keep_registered"/>
  </maintaingoalref>
  <achievegoal name="ship_goods">
  </achievegoal>
</goals>

<plans>
  <plan name="handle_delivery_request">
    <body>new HandleDeliveryRequestPlan()</body>
    <trigger>
      <messageevent ref="deliver_goods"/>
    </trigger>
  </plan>
  <plan name="ship_goods">
    <body>new ShipGoodsPlan()</body>
    <trigger>
      <goal ref="ship_goods"/>
    </trigger>
  </plan>
</plans>

<events>
  <messageevent name="deliver_goods" type="fipa" direction="receive">
    <parameter name="performative" class="String" direction="fixed">
      <value>SFipa.REQUEST</value>
    </parameter>
  </messageevent>
</events>
```

Figure 5.8 – Snippet of generated Jadex XML code

JACK Code Snippet:

```
agent Retailer extends Agent
{
  #uses plan SendOrderMessage;
  #uses plan DoShipment;
  ...
  Retailer(String name)
  {
    super(name);
  }
}
```

Figure 5.9 – Snippet of generated JACK code
5.3.6 System Documentation

As well as assisting users with the process of defining and developing a multi-agent system, a development support tool should provide facilities to assist with documentation as development progresses. NUMAP’s support tool allows the data entered into the system to be printed, providing simple user documentation.

A PDF file is generated, which collects the information about the elements from each phase into a single document. An example snippet of the documentation that is generated (for GBRAM agents) is shown in Figure 5.10.

<table>
<thead>
<tr>
<th>Agent</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Customer</td>
<td>Customer Agent, Responsible for placing orders with retailers.</td>
</tr>
<tr>
<td>Type</td>
<td>Agent</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Agent</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Retailer</td>
<td>Retailer Agent, Handles customer orders and optimises stock from suppliers.</td>
</tr>
<tr>
<td>Type</td>
<td>Agent</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.10 – Snippet of generated PDF documentation

5.3.7 Iterative Development and Dependency Tracking

Iterative development within the support tool is supported by two main features: versioning and dependency tracking.

Versioning provides the ability to maintain several concurrent versions of the design and to highlight differences between versions. When performing iterative design, each iteration of the design can be assigned its own version number. A text-based report can be generated to highlight the changes that have been made between versions, to assist with tracking changes and indicating where code changes are required.

Additionally, the support tool can track the dependencies between the various elements within NUMAP modules, using the schemes outlined in Chapter 4.
When this option is selected, a text file is created listing the elements that are affected by the change.

5.4  Design and Implementation of the NUMAP Support Tool

This section presents the details of the design and implementation of the NUMAP support tool.

The support tool is a complex software system, with a large feature-set. The software is comprised of six modules, divided into 44 packages, and containing 271 Java classes. Due to this complexity, the complete details of the implementation will not be presented in this chapter. However, an overview of the support tool, its design and structure will be presented.

![Diagram showing the components of the NUMAP Support Tool]

**Figure 5.11 – A core component provides shared code for all phases**

5.4.1  NUMAP Support Tool

The NUMAP Support tool is divided into a number of components, including one “Core” component and a number of phase components. For the base NUMAP process, there are phase components for Requirements Elicitation, Analysis,
Organisational Design, Agent Design and Code Generation. A high-level component diagram is shown in Fig 5.11.

The Core component includes common code that is used by each phase component to simplify implementation. This Core component can be divided into a number of subcomponents, as shown in Fig 5.12.

The CoreXML subcomponent assists with XML persistence, the CoreForms and CorePanel subcomponents provide classes which can be extended to create new UI elements, the CoreModel subcomponent provides abstract classes to store a data model for a process concept, the CoreCodeGen subcomponent provides classes to assist with code generation, and the Utility subcomponent provides miscellaneous functionality, such as collection management and ID number generation.

![Component Diagram]

Figure 5.12 – The core component is comprised of a number of subcomponents

5.4.2 Technologies

The NUMAP support tool has been implemented in Java, and is therefore usable on a wide variety of software platforms. This is desirable, as agent runtime
environments such as Jadex [39] and JACK [65] also have cross-platform operating system support.

XML has been used widely in the support tool. Persistence of model objects is done using an XML file format, and XML is also used in code generation for the Jadex runtime environment. XML functionality for the support tool has been provided by the Java dom4j library [148]. This allows XML files to be used within the support tool.

Support for printing and generating PDF documentation is provided by the iText library [149], which allows PDF documents to be created to display the details of concepts modelled in the process. Individual portions of the document can be built-up by implementing Java classes that extract information from the process model and use that information to generate document elements.

5.4.3 Process Model

As data is entered into the NUMAP tool, an internal process model is created. The process model is stored within the tool in an object that extends the AbstractProcessModel class. The process model class defines the data that will be stored for each phase of the process. This is done by referencing a number of individual phase model classes.

The standard NUMAP phase selection is defined in the DefaultProcessModel class. This defines phase models for Requirements Elicitation (ReqModel), Analysis (AnalysisModel), Organisational Design (OrgDesignModel), and Agent Design (AgentDesignModel). Each of these phase models extends the AbstractPhaseModel class (see Fig. 5.13).

No data has to be stored for the code generation phase of the process, because it simply takes advantage of data from the agent design phase, so no phase model is created for this element of the process.
Each phase model contains instance variables which store the data relevant to the phase. These variables store the information about the concepts for each phase. Concepts (such as goals or agents) are defined in classes that extend the `ModelConcept` abstract class. Each phase model contains lists that store the model concepts for the phase. For example, the default Requirements Elicitation phase model contains lists of actions (`ReqAction` objects), agents (`ReqAgent` objects) and goals (`ReqGoal` objects), see Fig 5.14.

Iterative development is supported by storing separate process models for each version increment.
5.4.4 Persistence

The NUMAP support tool allows data for the current project to be saved in a flexible XML [150] format. XML was chosen because it is a well-supported data format with Java libraries available to assist in its use. The use of a standard data format such as XML also opens the possibility of NUMAP project files being read by third-party tools.

The XML file that is created when saving NUMAP project files is based on a hierarchical format. The top level of the file format contains a number of process models. Each of these models represents a “version snapshot” of the project. The most recent process model represents the project in its current state.

Each of the saved process models consists of a number of phase models, which in-turn contain XML tags representing the process elements that are defined for that phase.

DOM4J [148] is used to allow the support tool to output XML files. This Java library provides support for creating an XML document as an object, then writing the document out to a file. Additionally, DOM4J allows for XML files to be parsed and converted into objects.

The choice to use a custom persistence model, rather than using an existing persistence technique such as Hibernate [151], or using Java serialization, allows for maximum flexibility and interoperability for saved files.

The NUMAP support tool provides an extensible framework for saving process models as XML files. A class extending an abstract SaveXML class exists for each ProcessModel class, and this then references a class extending PhaseXMLMarshal for each phase. The phase XML marshalling class is then responsible for converting the phase model into a DOM4J XML element, using a marshal method.
An example file for a skeleton NUMAP model (with one agent defined in requirements and no other data entered) is shown in Fig 5.15.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<model>
  <version>1</version>
  <reqModel>
    <agent>
      <id>ID1</id>
      <name>Customer</name>
      <type>0</type>
      <description>Customer Agent</description>
    </agent>
  </reqModel>
  <analysisModel/>
  <organisationalDesignModel/>
  <agentDesignModel/>
</model>
```

**Figure 5.15 - XML File Structure**

The relevant *PhaseXMLMarshal* class is also responsible for loading data from the XML file. An unmarshal method is defined for each class; it loads the data from a DOM4J XML element and converts the data into a process phase model object.

Implementation of the persistence code for new concepts is simplified by keeping common code, such as that for converting commonly used data types into XML tags and search operations, in the core module.

In summary, persistence occurs in the following manner:

- The ProcessModel component has a class extending *SaveXML*. An instance of this is called to convert the model to an XML document.
- The above instances then, in turn, calls the relevant class extending *PhaseXMLMarshal* for each phase.
- The *PhaseXMLMarshal* class then, in turn persists each process element defined in the phase by calling the relevant XML persistence classes for each concept.
5.4.5 Form-based GUI

NUMAP concepts are described by completing a series of forms. Each concept has its own distinct form, which lists the attributes that need to be defined to fully describe that concept. In the NUMAP support tool, forms are implemented as Java Swing [152] JPanels. All forms extend an abstract ProcessToolForm class which defines the basic controls and behaviour for forms in the process tool.

Forms are grouped into packages representing individual NUMAP phases. Within each of these packages there is one form for each of the concepts defined in the phase. Implementing the form for a new concept involves creating a new subclass of ProcessToolForm (and an associated embedded JPanel) for that concept.

The NUMAP support tool also provides a number of support classes for creating forms for new concepts. Classes for handling list data and combo-boxes are provided to simplify the process of creating new forms.

Forms access their relevant phase model directly to update the process model with data that is entered.

5.4.6 Process Guides

The NUMAP support tool provides functionality to assist users with the transition between phases of the process. This functionality takes the form of process guides. Process guides are used to simplify the act of moving from one phase to the next by filling in relevant data in subsequent phases of the process based on previously entered data.

The use of process guides is optional, but provides benefits if they are used within each phase of the process. The built-in NUMAP guides allow the user to select whether they wish to automatically fill in as much information as possible for the next phase based on information from the previous phase, or if desirable
for any reason, to manually map concepts from previous phases to subsequent phases, or to start the next phase with a blank slate.

Process guides are specifically implemented for a transition between phases. If a new phase module is to be added to the support tool, a new process guide must be specifically written for its transitions. The NUMAP support tool provides a template for creating custom process guides.

5.4.7 Diagrams

While forms are the mechanism for entering data into the NUMAP support tool, diagrams can be used to provide a visual representation of the relationships between different elements. Each phase in NUMAP can have its own specific diagrams showing the relationships between the elements in that phase.

An abstract diagram class, called ProcessDiagram has been created. Custom diagrams in NUMAP extend this class. Diagrams have access to their relevant phase model and can use this to access the required data. JGraph [147] is used for generating diagrams within the support tool.

A number of NUMAP diagrams have been created as a proof-of-concept in the requirements phase of the process.

5.4.8 Modularity

Modularity was a high priority in the design of the NUMAP support tool, in order to allow replacement process phases to be used with a minimum of effort.

The primary technique used for achieving modularity within NUMAP was the use of Java packages to separate the code for individual modules. A separate software component exists for each process phase, containing packages for the phase model, XML persistence, diagrams and forms.
Guides are separated into their own packages. They do not share the package with any phase, because guides represent a process transition, rather than a phase per se.

The individual process phases may be packaged into JAR archives [153]. All sub-packages in the hierarchy for the module can be packaged together, allowing the process phase to be treated as a pluggable module.

Presently, changes to the selection of modules require recompilation of the process tool’s code. This will be changed in the future to allow the used modules to be changed without the need for recompilation. This will be achieved by using Java Bean Shell [154]. Bean Shell allows code snippets to be sourced from a text file and executed at run-time, allowing the code that initialises the process model, process forms and process guides to be altered dynamically, and providing a mechanism to customise the process without the need for recompilation.

5.4.9 Document Generation

Design documentation can be automatically generated by the support tool. The documents provide a summary of the design in an easily-readable format. Documents are divided into chapters, one for each phase of the process. Each of the elements defined in the support tool is presented in a table, with any textual comments that have been entered for that element listed below the table.

A documentation module exists for each phase of the process, and is implemented as part of that phase’s module. When implementing a new custom phase in the NUMAP support tool, a new documentation module can to be implemented, using iText [149] to generate PDF files. Utility classes are provided to assist with this task.
Document generation is performed by selecting “Generate Documentation” from the NUMAP main menu. The user selects a path for the resulting PDF file, which is automatically generated using the iText[149] PDF creation library.

5.4.10 Iterative Development and Maintenance

Support for iterative development is common in large-scale software development [155], so it is desirable for the NUMAP process to provide support for this practice. In designing the NUMAP support tool, two decisions were made to assist with iterative development: support for versioning and the ability to track how changes to one element in a phase affects other related elements both in that phase and in subsequent phases.

Dependency tracking relies on the dependencies that are defined within the concept classes for each NUMAP phase. The abstract ModelConcept class contains a method called getDependencies(), which can be overridden to return the dependencies for an element, that is, the other elements on which it depends.

There are two kinds of dependencies, intra-phase dependencies, and inter-phase dependencies. Intra-phase dependencies are those dependencies that occur within a phase, for example in the GBRAM-based requirements phase, the dependencies of a goal would include its subgoals and scenarios.

Inter-phase dependencies are those dependencies that occur across phases. These are defined when using a guide to transition between phases. By adding elements to the subsequent phase based on those in the current phase, a link is created between the two. This allows dependencies to be highlighted between phases, so impacts on later phases can be identified when considering a change to an element in an early phase, such as Requirements or Analysis.

The NUMAP support tool can highlight both the direct dependencies for an element and the dependency hierarchy for that element. This is done by
generating a dependency report (via the File menu). This creates a text file that lists the dependencies, allowing the impact of any change to be assessed by the developer.

5.5 Difficulties, solutions and alternatives

A number of alternative approaches were considered when implementing the NUMAP support tool. This section will describe these alternatives, along with difficulties that were encountered and the solutions that were developed to address them.

5.5.1 Data Entry

Selection of appropriate input schemes was of vital importance when designing the support tool. One of the requirements identified at an early stage was to allow rapid data entry of concepts by the user. A large amount of data needs to be specified when using the support tool, so the choice of data entry scheme greatly affects usability of the tool.

Two alternative choices for data entry were identified: a graphical diagram-based system of data entry and a form-based user interface using simple user interface elements. Each of these choices has distinct advantages and disadvantages.

A graphical system for entering elements into the support tool was considered. Such a system would use a number of diagrams as a framework to specify the design. Elements would be entered as nodes in a diagram, and relationships specified using connections between nodes. Such a system is used to enter UML [156] data in Rational Rose [157].

The advantage of such a system is that a pictorial representation of the system is built up as data is entered, allowing relationships between elements to be visualised. The disadvantage of such a system lies in the complexity of its implementation. Diagrams cannot easily be implemented using the core Java
libraries, so an external Java library, such as JGraph [147] is needed. This requires familiarity with the library for implementation of custom diagrams, adding to the complexity of implementing new modules for NUMAP.

An alternative approach is to allow elements to be specified via a form-based GUI using standard Java Swing [152] elements. This is the approach that was taken when designing the NUMAP support tool. Such an approach is relatively simple to implement, as it requires no knowledge of external libraries. It also allows rapid entry of data via the form-based user interface.

Visualisation of information is still desirable, so the support tool allows diagrams to be implemented using JGraph. These diagrams are optional, and separate to the data entry system. They allow illustration of relationships between concepts.

A system for automatically generating user interface forms via a configuration file was also considered. This would remove the need for programming when customising the process. However, it was decided that the time required to implement such a system -- or adopt an existing user interface management system for the task -- would be excessive. This was deemed out of scope for the current project, but may be considered in future revisions of the support tool.

One final difficulty faced in creating system forms for organisational design was the specification of Interaction Protocols. The initial strategy for implementing this functionality in the support tool was to create a tool for drawing interaction protocol diagrams within the software.

However, it was decided that a more effective approach would be to have the diagram created in an external tool that supports the creation of sequence diagrams. This allows the user to use tools with which they are familiar, and avoids duplication of functionality that already exist in other programs.

Any program capable of specifying interaction protocols, could be used, for example, Rational Rose [157]. These diagrams can then be referenced from
within the NUMAP support tool when specifying the Interaction Protocol element.

5.5.2 Modularity

When designing the NUMAP support tool, it was decided that the code should be separated into components based on the phases of the process, in order to satisfy the requirement of modularity. As a result, classes specific to individual phases are separated into distinct components, and all common code is grouped within a common “core” module. The core module is lightweight, and serves to provide the basic GUI and framework of classes on which the components of each phase are built.

An earlier version of the NUMAP support tool divided code into components based on functionality, rather than phases. For example, separate components existed for code generation, document generation, and for GUI forms. This provided high cohesion, but did not satisfy modularity requirements.

The decision to divide code into components based on phases assists with the extensibility of the support tool. The core component provides support classes for developing new phases, or extending existing ones. These support classes have been used to develop the standard set of design phases that exist within NUMAP.

5.5.3 Guides

A feature named “guides” has been implemented in NUMAP in order to reduce the work required by the user when transitioning between phases of the process. Guides assist by reducing the amount of data that must be entered.

When moving from one phase to another, the user may choose to either: select elements in the current phase that directly map to those in the subsequent phase, and auto-fill data for the successor element where possible; to automatically create elements in the subsequent phase based on the current
phase (a fully automated process); or to start with a “blank slate” and enter all concepts for the new phase manually.

Where elements from one phase are used as the basis of an element in a subsequent phase, a link is created between the two elements. This link is stored as a predecessor association in the model class representing the element. These links may also be specified manually by the developer.

Allowing the user a choice from a number of approaches to transition between phases provides flexibility for developers. They may choose to automatically fill in data for subsequent phases, to take a more managed approach by manually specifying mappings, or to start from scratch.

Guides have been implemented for the standard NUMAP process. For custom processes, guides would need to be developed with the full process in mind, rather than just for a custom phase. The functionality of a guide will differ based on the pair of phases with which it is associated.

Replacing a phase in the process will require any guides associated with that phase to be replaced or customised, however, in practice, this is a lesser concern, as the phases that are typically replaced or modified are the Requirements Elicitation and Agent Design phases (the first phase and the final design phase). As a result, swapping out one module for another would typically only involve replacement of a single guide.

5.5.4 Code Generation

Code generation is implemented within the support tool via a separate code generation component for each platform that is supported. An abstract CodeGeneration class is provided in the core component, along with a number of utility classes for handling NUMAP models for code generation and for file creation. Actual implementation of an individual code generation component may be done in any manner, with the developer free to choose the best
approach, depending on the format of the code to be generated (XML, Java, C++, etc).

The support tool allows the target platform for to be selected upon code generation. Presently, Jadex and JACK are supported, allowing code templates to be generated for these languages. Support for other agent environments can be provided by implementing a code generation module for them, then referencing it from the code generation form.

A model transformation [158] approach, such as QVT [159], could be used for code generation. Such an approach would allow a more formal technique to be used for generating code from the NUMAP model. QVT uses a “Relations Language” to define mappings between the source model and the target model. In the context of NUMAP, this would allow elements in the design model to be mapped to code fragments. NUMAP concepts would be equated to the desired code using the Relations Language, and QVT would use a “transformation machine” to assess equalities between the source and these code fragments to generate the output source code (the target model).

As a proof of concept, a simpler approach was selected for the initial version of the support tool. Code template generation has been manually coded, with output files for each implementation environment generated based upon the data stored in the NUMAP model.

### 5.6 Novel and Notable Aspects

There are three key novel aspects of the NUMAP support tool

1. The NUMAP support tool can be modified to support new modules as required. In order to support this, common “support” code has been centralised into an independent module and can be used for implementing extensions to the process. This supports the NUMAP
philosophy of having a base process which can be expanded to support specific requirements.

2. Generation of code templates has been fully implemented for the core process. Code may be created for Jadex and JACK. Code generation for these languages can be altered when extending the process, and new languages may be implemented using the provided support classes.

3. “Guides” allow transitions between phases to be assisted, tracking dependency mappings between phases. Dependency reports can be generated for any process element, providing a text-based list of elements that have a dependency relationship on it, either within a phase or between phases.
Chapter 6  Evaluation

6.1  Introduction

The aim of this work has been to create a usable, practical software development process that allows multiagent systems to be developed at a level of abstraction that takes advantage of the close mapping between BDI concepts and the natural mental notions used when people reason about actions. The concepts surrounding “human action” should be central to the process.

In order to evaluate whether the process meets this aim, it is necessary to ensure it adequately models all concepts required in BDI agent design. The model must cover both the internal reasoning of agents and the collaborative aspects of multiagent system design. Additionally, the process must be usable, complete and consistent.

6.2  Selection of an Evaluation Methodology

Evaluating a software process is not a simple task, particularly when resources are limited. Generally evaluations of software processes fall into one of three main categories: formal experiments, quantitative case studies and feature analysis evaluations [160].

While a large-scale experiment or even a quantitative case study comparing multiple processes would be an ideal way of evaluating a newly developed software process, this approach has several drawbacks when resources are limited.

To begin with, the selection of system to be implemented presents difficulties. Some systems are naturally more challenging to implement than others, and some play to the advantages of a particular design process more than others.
Therefore, for the comparison to be a truly fair one, a number of different problems should be implemented using each of the processes.

Additionally, some developers are naturally more comfortable with designing and implementing multiagent systems than others, so a comparison should include a number of different developers using each process.

Meeting these requirements would involve considerable resources. A large-scale experiment consisting of multiple developers implementing multiple projects using a range of development processes would require resources that simply were not available during this research.

A smaller-scale case study is an alternative, involving a single developer implementing a single problem using ad-hoc development, and a number of different processes for comparison.

However, several problems are evident with such a case study. First, familiarity with a process plays a large part in the developer's success in using it. This factor can quite easily outweigh others during the evaluation. Second, once the problem is successfully implemented in one process, the knowledge of that solution will influence future implementations using other processes. It is difficult to avoid ending up with a similar solution each time, with the design being guided more by memory of prior implementation, rather than by the process itself.

Due to these factors, it was decided to use a feature-based evaluation technique rather than one based on a specific development task. This feature-based evaluation is the subject of the remainder of this chapter.

It is recognised that this approach has its drawbacks. Expectancy bias may lead to preference for a process that is more familiar, and for selection of features that are of particular relevance to the work being presented. This has been mitigated by the use of existing peer-reviewed literature as the basis for selecting features for evaluation. However, it is acknowledged that evaluation
by independent subjects experienced in the design of multi-agent systems should be a focus of future work.

6.3 Feature analysis-based evaluation

A feature analysis was determined to be the most objective way to conduct an evaluation of multiagent development processes. The feature analysis establishes the completeness of the process, by assessing it according to a set of targets that an ideal process should meet.

A number of feature-based evaluation methodologies have been proposed specifically for evaluating agent-oriented development processes [161-164]. These methodologies define criteria upon which to assess the quality and completeness of a given development process.

Sturm and Shehory describe an evaluation framework for agent-oriented software processes [161] that addresses the comprehensiveness and quality of a process based on a number of feature criteria.

These criteria are divided into four categories: “Concepts and Properties”, “Notations and modelling techniques”, “Processes”, and “Pragmatics”.

Features are assessed and assigned a numerical score between 0 and 7, depending on how well the process to be analysed addresses the requirements of the metric. This provides a clearly defined qualitative analysis of processes that allows both the individual features and overall quality of a process to be compared in a simple manner.

A number of other similar techniques for feature analysis of agent-oriented processes have also been proposed. Cernuzzi et al [162] used the Goal Question Metric to devise evaluation criteria for agent-oriented processes. This framework proposes an evaluation criteria similar to that of Sturm and Shehory.
in that it provides a collection of attributes to be analysed, and criteria for assigning a numerical value to each feature.

The main difference lies in the selection of criteria, with an attribute tree being defined that specifies a number of features, grouped according to whether they are internal attributes of the agent, interaction attributes for the system, or “other process requirements” that an agent modelling method should offer.

A more comprehensive and wide-ranging feature analysis is specified by Tran, Low and Williams [163]. This provides a range of criteria, grouped according to whether they are process related, technique related, model related, or associated with support features of the process. Unlike the two aforementioned techniques, the Tran, Low and Williams approach does not assign a precise numerical value to each feature, instead using a more descriptive approach for analysing them.

Dam and Winikoff [164] developed a framework for the comparison of agent-oriented methodologies. This framework is based upon a questionnaire to be filled in by process authors, describing how well the process meets a number of criteria, and the completeness of the process itself. This questionnaire provides a concise summary for comparing processes, but does not assign a numerical rating to the process to allow comparison in a more quantitative fashion.

While these four evaluation methods have different outputs, they all share the same basic methodology of performing a feature analysis on the processes to be compared. In order to create an evaluation framework for comparing NUMAP to existing processes, we will compile a list of the most common shared features between evaluation frameworks.

As the proposed evaluation scheme is to be focused on evaluating newly-developed processes, we will not select features that are only relevant to long-established processes, such as those to do with user-base and historical usage.
6.4 Selection of Features

Selecting the features to be used for evaluation is critical in ensuring the final framework adequately assesses the quality of the NUMAP process. The evaluation methodologies described above each use a unique range of features, but there is a degree of commonality between them.

In creating a new evaluation framework, the intent is to identify the most important characteristics, and provide an even spread of features from different categories. To achieve this, the features from each of the four evaluation schemes described above [161-164] were listed, and the features existing in at least three of the schemes were highlighted. Some features from the resulting list were then merged when they covered very similar concepts, and the resulting list of features was divided into categories, creating several feature groups.

The remaining features were assessed based on their perceived importance, and by their degree of overlap with features already selected. Additional features were selected for inclusion when they covered areas that had not been previously addressed, and a number of additional features were introduced, consistent with the goals identified in Chapter 2. This assisted in providing an even division of features across categories. The resulting feature list provides representative criteria for evaluating development processes for BDI multiagent systems.

6.5 Overview of Features for Evaluation

The following sections describe the features included in each of the categories. An explanation of the importance of each feature is described, along with an explanation of how it will be assessed.
6.5.1 Core Agent Properties

The first feature category is used to assess the ability of the design process to model the core properties associated with software agents. For the process to be considered suitable for developing multiagent systems, it needs to provide mechanisms for modelling software components that support each of these properties.

The key property associated with agents is **autonomy**. The ability to execute without supervision or user intervention is generally considered essential for a software component to be considered an “agent” [36-38]. This concept is considered key to agent development, and was included in all four of the existing evaluation frameworks described above.

Agents should pursue their goals without user intervention being required, and should consider alternative courses of action when necessary. Such consideration of alternatives should be based on the state of the agent and the state of its environment. For a development process to rate highly in this category, it needs to model concepts that support such autonomous execution.

The development process should support **reactiveness** within agents. Agents should be responsive to external events, performing actions in response to external stimuli without delay [37, 38]. Ideally, an agent should be able to respond to predefined events without the need for any time-consuming deliberation. Predefined mappings between actions and events can result in immediate action, whereas response to non-predefined events can take significantly longer.

Predefinition of urgent events and appropriate responses to them allows an immediate agent to take immediate action in time-sensitive situations. Reactiveness is included in all four of the existing evaluation frameworks. For a development process to be considered to address this feature, it must provide a mechanism for specifying this reactiveness within agents.
The third of the core agent properties is **proactiveness**. An agent should select and pursue goals in order to fulfil its role in the system. Goals for an agent should be clearly expressed within the process, and the process should be flexible enough to allow the definition of a variety of goal types. This feature is included in each of the above evaluation frameworks, either as “proactiveness” or “deliberative behaviour”. For a process to satisfy this, it must include mechanisms for modelling proactive behaviour.

These three features, autonomy, reactivity and proactiveness form the “core agent properties” category of the evaluation. These properties are considered essential for a process to be considered suitable for multiagent system development.

### 6.5.2 Agent Modelling Concepts

The second feature category assesses the compliance of the process to the modelling concepts that are generally used for describing multiagent systems. For a process to rate highly in each of these categories, it needs to provide mechanisms for specifying each of these concepts in detail.

The process needs to allow **agent** models to be defined, providing a clear means for describing the agents in the system. Two of the four evaluation frameworks explicitly include a feature for agent specification, or for “agent orientation”. For this feature to be successfully addressed, agents must be specifically modelled within the system, along with specification of their intended behaviour, and details of the services that they provide.

The process must also adequately model the mental attitudes of agents. For a process to adequately address BDI agents, it is evident that the mental notions key to this model are adequately addressed. All four evaluation frameworks explicitly specify these as features to be evaluated.
Agent beliefs need to be specified, adequately modelling the information that the agent believes to be true about its environment, other agents, and itself. The possible desires, or goals, to be pursued by the agent are defined, along with the conditions upon which those goals may be triggered. These triggers should be defined for both achievement goals, where a goal is pursued to bring about a new state in the system, and for maintenance goals, where a goal attempts to preserve a pre-existing state.

Additionally, the means by which an agent pursues its goals need to be specified. A mechanism for mapping intentions to desires is needed, clearly defining which intentions may be used for realising each desire. These intentions are commonly termed plans.

Larger multiagent systems require agents to be structured into groups, allowing for modular design of the system, and providing structure to its inter-agent design. All of the above evaluation frameworks include a feature for assessing how well the process defines the organisations in the system, or how the process addresses modularity within the agent specifications. To rate highly for this metric, it must be possible to group agents into organisations, in which each plays a role in achieving the organisation’s common purpose.

In order for agents to effectively collaborate, their communications must be well-defined. A good process must allow for the specification of interaction protocols between agents, defining the admissible communication patterns. Three of the four evaluation frameworks specify this as a required feature. For a process to rate highly for this metric, interaction protocols must be clearly and specifically modelled, defining message formats and patterns of communication.

Agents exist in dynamic, changing environments, where evaluation of alternatives is key to an agent’s effective performance. When an agent requests a service from another entity, multiple providers may be available for that service with differing degrees of reliability. When an agent attempts to pursue a goal, multiple plans may be available to satisfy that goal. As noted by Tran, Low
and Williams, deliberation, inferential ability, and adaptability are important aspects of multiagent systems. An agent design process must therefore adequately model the concept of selection rules, both for plans and for agent services.

Together, these features form the “agent modelling” category. For a process to rate highly in this category, it must adequately model the following concepts: agent, belief, goal, plan, organisation, protocol and selection rules.

6.5.3 Completeness and Clarity

The third feature category assesses the completeness and clarity of the process. It measures whether the process adequately covers the entire process of developing multiagent systems, whether is clear and consistent, and whether it adequately addresses the unique concerns faced by highly concurrent systems featuring autonomous, cooperative entities.

A high-ranking process should cover the entire gamut of software development, from requirements elicitation through to maintenance and testing. Each phase of the process should be defined in detail, describing the procedures that need to be followed. The entire life cycle of software development must be addressed. This feature is included in three of the four above evaluation frameworks.

The process should have a clearly specified modelling syntax. All details required for each concept used within the model should be clearly defined. Without clear models, it is difficult for a developer to correctly follow the process, and errors or omissions may occur. The preciseness or clarity of models is included in the feature analysis for all of the above frameworks.

The models that are created should be consistent. Consistency should be enforced within each stage, between phases, and with any finished product created as a result of the process. It should preferably be supported by software
tools, allowing the models to be analysed. This feature is addressed in two of the above evaluation frameworks.

The process needs to effectively model cooperation between agents. Within a multiagent system, goals may often only be achieved through teamwork. Mechanisms should be available within the process to model cooperation between agents to achieve such common goals. Three of the four evaluation frameworks assess processes for their support of teamwork or cooperation.

The process should also take into account the concurrency inherent to multiagent systems. Techniques for managing concurrency are needed within the process. This is addressed by two of the above evaluation frameworks.

For an agent design process to be applicable to modern design approaches such as agile programming, it must support iterative design. Therefore, an additional assessment criterion is added: iterative support.

The “completeness and clarity” category therefore includes the following features: life cycle, clear models, consistency, cooperation, concurrency and iterative support.

6.5.4 Usage and Practicality

The final feature category assesses a process for usability and practicality. It assesses how readily the process may be used in practice to a range of application domains and project sizes. A high-ranking process must have a high degree of usability for developers, and there must be mechanisms to tailor the process.

The process should be applicable to a number of application domains. It should not be specific to a particular industry, development technique, or implementation platform. It must have a high degree of domain applicability across real-world domains. This feature is addressed by three of the four above evaluation schemes.
The process should be able to scale to handle both small problems and complex systems. Ideally, it should be possibly to simplify, to provide a lightweight version for less complex applications. **Scalability** is included in three of the four above evaluation frameworks.

The process must assist the developer with handling large projects through means of **abstraction** and complexity management. Models must be defined in a hierarchical manner, so developers can obtain a high-level view of the system, while also being able to “drill down” to find more specific detail about entities in the system. The models within each phase should only be relevant to that phase – details pertinent to other phases should be omitted. This is included in three of the above schemes.

**Usability** is clearly a priority for a development process. The process should be as simple as possible for developers to follow. As well as clarity and a straightforward methodology, the process should provide tools and other support mechanisms to allow the developer to more easily follow the process. Usability is addressed by two of the above schemes.

The process should be easy to learn for new users. It should also be simple for novices with little expertise in multiagent systems to adopt. Users should not need to understand complex mathematical models or formal logics. The process should have a high degree of **learnability**. This feature (learnability, or required expertise) is included in two of the above schemes.

Tools and techniques for the **reuse of models** between projects should be provided by the process. If possible, the process should specifically include concepts encouraging reuse of components. This feature is addressed by two of the above evaluation frameworks.

A design process must be flexible in its support for varying types of agent. Tran, Low and Williams refer to this as “agent nature”. An agent system is more flexible and practical if it supports **heterogeneous agents**.
The “usage and practicality” category therefore includes the following features: *domain applicability, scalability, abstraction, usability, learnability, reuse of models, and heterogeneous agents.*

### 6.5.5 Metrics

As with the evaluation scheme defined by Sturm and Shehory [161], each feature will be allocated a score out of 7, measuring the degree to which it meets the requirements outlined above.

The scores to be assigned to each process are based on the ranking system introduced in the Sturm and Shehory paper. Our description of the ratings is as follows:

- A rating of “1” indicates that the feature is not addressed by the process.
- A rating of “2” indicates a reference to the feature.
- A rating of “3” indicates limited coverage of the feature.
- A rating of “4” indicates major issues, or a process lacking some aspects of the feature.
- A rating of “5” indicates some small, but important, missing aspects.
- A rating of “6” indicates some small issues.
- A rating of “7” indicates the feature is fully included.

### 6.6 Feature analysis of NUMAP

In this section, evaluation framework described above is used to assess the following processes:

- NUMAP
- Gaia (the original process, defined in [20]).
- Prometheus [22].

The NUMAP process is addressed first. Each feature is listed, with a description of how well the process addresses that feature. A rating for the feature is then provided.
6.6.1 Core Agent Properties

*Autonomy* – Autonomy is core to agent design in NUMAP. Agents are assumed to be autonomous, and selection rules are included to model consideration of alternative plans and service providers. Agents monitor their environment through sensors and effectors, and respond through goals and reactions. Services allow agents to dynamically collaborate. Rating: 7.

*Reactiveness* – External events can be responded to immediately via reactions, which are explicitly modelled in NUMAP. Events that need an immediate response can be defined, and a “hard-wired” response specified, allowing the agent to respond in a timely fashion without the need to engage in a temporally costly deliberation phase. Rating: 7.

*Proactiveness* – The proactive nature of agent systems is a central part of NUMAP. Goals and Plans may be defined in detail, and the mapping between these concepts is defined. Deliberation rules for selecting which plan meets the requirements of which goal are also explicitly modelled. Both achievement and maintenance goal types may be specified within the process. Rating: 7.

Thus, the NUMAP process, being explicitly designed for BDI agents, fully addresses all core agent properties.

6.6.2 Agent Modelling Concepts

*Agent* – Agents form a central part of the NUMAP process. This extends from the agents and stakeholders defined in the GBRAM requirements module, to the social-level description of agents in organisational design, through to the detailed specification of agents in the agent design phase. Rating: 7.

*Belief* – NUMAP supports the definition of beliefs for agents, the specification of belief types and the ability to provide descriptive notes for the handling of each belief. Rating: 7.
**Goal** – NUMAP supports definition of goals of different types, as well as defining deliberation selection rules for managing those goals. Rating: 7.

**Plan** – Plans may be defined in NUMAP, and the goals for which they are appropriate defined. Additionally, plans may be triggered by a reaction, allowing immediate, reactive adoption of intentions. Rating: 7.

**Organisation** – NUMAP provides an organisational design phase to define the relationships between agents, and provides the ability to group agents according to organisations during this phase. However, advanced features such as dynamic runtime restructuring of agents or explicit organisational rule modelling are not addressed in the initial version of NUMAP. Rating: 5

**Protocol** – Interaction protocols may be defined during the organisational design phase of NUMAP, using FIPA [130] standard Agent UML [72] modelling. Additionally, messages may be defined during this phase, for use with interaction protocols. Rating: 7


NUMAP largely addresses the required agent modelling concepts, albeit with some limitations in organisational modelling. Enhanced organisational modelling would be desirable for a future extension to the process.

### 6.6.3 Completeness and Clarity

**Lifecycle coverage** – NUMAP covers all aspects of the agent development lifecycle from requirements to implementation. However, maintenance and testing are deemed out of scope at this stage. Rating: 5

**Clear Models** – Models are defined clearly (see Chapter 4), with detail provided about the exact data types required for specifying concepts. However, to retain flexibility, the exact format of each field has not been enforced. Rating: 6
**Consistency** – The support tool provides dependency tracking for encouraging consistency between iterations of development. “Guides” for transitioning between phases encourage consistency between phases. Dependency tracking can also be used for highlighting potential inconsistencies. However, further implementation could be provided within the support tool to detect potential inconsistencies in the models. Rating: 5.

**Cooperation** – The design of NUMAP encourages the development of cooperative agent systems. Agents use the services of other agents to aggregate functionality and create the overall behaviour of the system. Organisations can be defined to group together cooperating agents. As noted above, further work could be done on extending the modelling of organisations. Rating: 6.

**Concurrency** – The concurrent nature of agent systems is effectively modelled during NUMAP’s organisational phase. Communication constructs such as Interaction Protocols model the asynchronous communication using Agent UML sequence diagrams. Rating: 6.

**Iterative Support** – NUMAP has been designed to support iteration through the process, and provides dependency tracking within its support tool to assist with this. The novel nature of its phase transitions allows dependencies to be tracked between phases, as well as within them. Rating: 6

NUMAP provides a strong degree of consistency and concurrency. However, additional support for implementation and testing, and for consistency checking within the support tool would be beneficial.

### 6.6.4 Usage and Practicality

**Domain applicability** – The process has been designed to be applied in a wide range of applications and domains. It is not tied to any particular field of multiagent systems, and its modular design even allows it to be extended to handle non-BDI agents. The code generation is extensible, and presently
supports two implementation platforms. However, it has not been specifically designed for open agent systems, where trust of agents is an issue. Rating: 6

*Scalability* – NUMAP has been designed to be easily followed for small systems, but also to include enough detail and structure to be used in development of larger applications. Additional organisational functionality could potentially improve large-scale implementations. Rating: 6

*Abstraction*– NUMAP is designed to offer hierarchical refinement of agents as the developer progresses through the stages of the process. As the developer progresses to subsequent phases, new detail and additional concepts are introduced. Rating: 6.

*Usability* – One of the prime concerns when developing the process was usability. The decision to develop a coarsely-grained modular process (rather than using a meta-methodology), the development of a support tool, and the flexibility of allowing different requirements modules to be developed were all decisions made to enhance the usability of the process. Additional user testing will be done in the future to further refine the usability of the NUMAP process. Rating: 6.

*Learnability* – The NUMAP support tool is designed to assist novices in following the process. Additionally, the modular requirements module allows for the definition of a different requirements phase that may better suit the expertise of particular project teams. In the future, it would be beneficial to have NUMAP formally evaluated by novices unfamiliar with multiagent systems. Rating: 6.

*Reuse of models* – NUMAP provides the concept of “capabilities” to encourage the reuse of models at the agent design phase. However, more focus could be placed on reuse within the support tool. Rating: 5.
**Heterogeneous agents** – NUMAP has been explicitly designed to support heterogeneous agents, and the modularity of its agent design phase allows for new agent design modules to be used for different agent types. Rating: 7.

NUMAP largely provides the features required by the “usage and practicality” category. However, further user testing and enhancements to the support tool are desirable.

### 6.7 Feature analysis of Gaia

The Gaia methodology [20] is one of the more prominent and influential design processes for developing multiagent systems. As such, it is included in this evaluation.

A feature analysis of the Gaia process has been undertaken by Sturm and Shehory, using the scheme described earlier in the chapter [161]. In evaluating the Gaia process, it would be sensible to take advantage of this existing evaluation, because it covers many of the same features as the evaluation presented in this thesis.

Gaia’s properties were assessed independently, then feature ratings from the above evaluation were considered where they were common to both assessments. Where there was no disagreement between assessments, these ratings were used. This is noted in the ratings, with the notation “as in S&S”. If a conflicting rating was found, this is also noted. Where no mention has been made of a rating, it was not specified in the above evaluation.

#### 6.7.1 Core Agent Properties

**Autonomy** – The Gaia methodology provides excellent support for autonomy through its definition of roles. Each role is defined in an autonomous fashion, with alternative communication paths being defined to allow the agent performing the role to respond autonomously to differing conditions. Rating (as in S&S): 7.
**Reactiveness** – Gaia provides some support for reactive behaviour, through the definition of “liveness expressions”, which may make reference to external events, and specify the actions to be taken when they occur. However, these are not expressed at a detailed level – there is no specific modelling of the agent's immediate response to events and triggers. Rating (as in S&S): 3.

**Proactiveness** – The Gaia “liveness expressions” do provide a good mechanism for modelling proactive behaviour. This feature is well-handled by Gaia. Rating (as in S&S): 7.

Gaia addresses the autonomous and proactive properties of agents, but lacks specific support for modelling reactivity.

### 6.7.2 Agent Modelling Concepts

**Agent** – Agents are central to Gaia. They are modelled in detail, including their roles, responsibilities and interactions. Rating: 7.

**Beliefs** – As noted in S&S, Gaia does not specifically model beliefs, goals or intentions. Internal knowledge representation for agents is not specified in any detail. External resources, and the agent's permissions to access them, are however modelled. Rating: 2.

**Goals** – The desires of agents are handled to some extent, through “liveness properties” and “safety properties”, however internal representation of goals, selection and reasoning are not covered in any detail. Rating: 3.

**Plan** – As Gaia is not focused on concrete design, there is little detail of plans. The agent's “liveness properties” do express some detail about what is to be achieved for each goal, but this is at an extremely high level. Rating: 2.

**Organisation** – Organisations within Gaia are defined in terms of an acquaintance model. The interactions between agent types are specified, but the focus is not on modularity, rather it is on seeing the system as a single
organisation. Organisations are static, and cannot be altered at runtime. Due to the limitations of the organisational modelling in Gaia, the scope for organisation-centric design is limited. Rating: 3.

Protocol – Gaia provides a specific interaction model for agents, defining protocols to be used for communication. However, these are presented in an abstracted manner, focusing on the nature of the interaction, rather than the precise sequence of steps required. This presents significant difficulties if interactions are to be implemented from these protocols. Rating: 4.


Gaia’s support for BDI agent modelling concepts is mixed. The concepts are addressed to some degree, but often in a non-specific manner that leaves their design open to interpretation by the developer. This is largely due to Gaia’s focus on creating a model of the system that is abstracted away from specific low-level design approaches.

6.7.3 Completeness and Clarity

Lifecycle coverage – Gaia only defines the analysis and high-level design phases of development. It has no detailed design element, and is not designed to model any specific agent design technique. This severely limits its ability to be used to develop real-world software. The process must be extended to support the desired implementation platform, and no guidelines are proposed on how to achieve this. Rating (as in S&S): 3.

Clear Models – Models for concepts explicitly specified by Gaia are clearly defined. Liveness and safety properties are presented clearly, with well-defined notation. However, the focus on early development leaves many aspects of the process without any definition at all, and many activities that need to be performed are only mentioned in passing. Rating (as in S&S): 5.
**Consistency** – Gaia’s use of temporal logic ensures models are consistent and correct. However, there are no automatic tools to assist in analysing the model, and achieving consistency with the implemented agent system may be difficult, due to the absence of low-level design. The “analysability” of Gaia was rated a “1” by S&S. However, this rating seems unfairly low, so a higher rating has been assigned in this evaluation. Rating: 4.

**Cooperation** – Gaia manages cooperation between agents using roles, and a well-defined interaction model. This is handled well, albeit with the limitations in the specification of interaction protocols that were mentioned above. Rating: 6.

**Concurrency** – Gaia defines agents as coarse-grained autonomous entities, with clear interaction patterns. However, the precise sequences of steps required for communication are not modelled, which will hamper implementation. Rating: 5.

**Iterative Support** – Gaia makes some reference to iteration, but no support is given for doing so. Rating: 2.

Gaia presents a clear means for modelling agents, with detailed notation provided for each of its internal models. However, the lack of lifecycle coverage, and the abstraction inherent in its interaction protocols hamper real-world implementations. As such, it receives mixed scores in the “completeness and clarity” feature category.

**6.7.4 Usage and Practicality**

**Domain applicability** – The authors of Gaia state that is applicable for the development of real-world applications, with the following characteristics: agents are coarse-grained in their functionality, the system pursues a global goal (without true conflict between agents), the system’s services are static, and
the system has less than 100 agent types [20]. Dynamic and open systems are not possible with Gaia.

However, within these constraints, the abstract nature of the Gaia process ensures it can be applied to a wide range of domains. As completeness is factored into the ratings in the preceding section, Gaia will not be penalised for the practical difficulties in applying it to specific agent systems, and a higher rating will be assigned than that by S&S. Rating: 5.

**Scalability** – As noted above, Gaia is suited for projects with less than 100 agents, and its lack of modularity and componentisation may hamper its use on large-scale projects. Rating (as in S&S): 4.

**Abstraction**– Gaia includes two phases: analysis and design. However, due to the high-level focus of the methodology, the complex, lower-level functionality is not specifically handled. It is at these final stages of development that abstraction and complexity management is most necessary. As such, rating this feature is difficult. In S&S, a rating of 1 is assigned for complexity management. However, a higher rating is assigned in this evaluation, in recognition of the clear separation within Gaia between agents, roles and protocols. Rating: 3.

**Usability** – Gaia provides a clear syntax for defining its models. However, the ultimate usability of the process lies in the ability to implement a system based on the created model, at its completion. This is difficult with Gaia, as the process does not cover the later stages of development. Rating: 3.

**Learnability** – Gaia’s clear syntax may assist with learnability. However, the lack of coverage of BDI concepts hampers the ability to consider design at this level of abstraction. External concepts, such as “liveness properties” and “safety properties” need to be understood to properly use the process. Rating: 3.

**Reuse of models** – Gaia does not provide any explicit guidelines for reuse. However, the use of roles for defining agent functionality may help in this regard. Rating: 3
**Heterogeneous agents** – Gaia does not provide a means of defining agents at a detailed level, whether homogeneous or heterogeneous. However, the lack of specificity could be seen as allowing flexibility in design. Rating: 2.

Gaia’s practicality is hampered by its abstract design and focus on the early phases of development. Actual implementation of systems using the process is a challenge, and the lack of coverage of BDI concepts makes it difficult to consider the design from this higher level of abstraction.

6.8 Feature analysis of Prometheus

A feature analysis of the Prometheus methodology is now presented. In using various processes throughout the course of this research, Prometheus was found to be one of the most practical and complete processes for developing multiagent systems. It is clearly specified in Padgam and Winikoff’s book on the process [22], so this specification was used as a basis for the evaluation.

Prometheus has been assessed by two of the above four evaluation papers, albeit not using specific numerical metrics. These evaluations were taken into account when rating each of the below categories, but the primary sources for the evaluation was the description of the process by Padgam and Winikoff.

6.8.1 Core Agent Properties

*Autonomy* – Prometheus offers comprehensive support for agent autonomy throughout the process. Agents are referred to as “automomous situated components”, and this reflects the way they are modelled. Goals, processes and plans are all clearly defined. Rating: 7.

*Reactiveness* – Reactiveness in Prometheus may be defined in terms of event descriptors. Percepts and messages can trigger events, resulting in a reactive response. Rating: 7.
**Proactiveness** – Prometheus defines agents in terms of “capabilities” that are proactive in nature. Goals, plans and beliefs are all adequately modelled, providing the ability to implement complex proactive behaviour. Rating: 7.

The Prometheus methodology provides excellent coverage of the agent concepts, due to its focus on modelling BDI multiagent systems. It fully addresses all core agent properties.

### 6.8.2 Agent Modelling Concepts

**Agent** – Prometheus includes agent types as a basic concept. Agents descriptors are defined during *architectural design* and are a key element of the methodology. Rating: 7.

**Belief** – Beliefs in Prometheus are specified using “data” descriptors. Defining this data is not a focus of Prometheus, and it is recommended they be defined using “some appropriate methodology”. Rating: 5.

**Goal** – Goals are used throughout Prometheus, and are identified from the earliest stages of the process. Refinement of goals is a key activity in the *system specification* phase of the process. Goals are a secondary concept during the *detailed design* phase, due to the close coupling between Prometheus and JACK (which implements goals as events), but they are referenced in capability and plan descriptors. Rating: 6.

**Plan** – Plans are explicitly defined in Prometheus using *plan descriptors*. Triggers, goals, procedures and failure conditions are defined in detail. Rating: 7.

**Organisation** – Organisational groupings of agents are not explicitly handled in Prometheus, however communications between agents are well defined. Rating: 4.
Protocol – Prometheus allows interaction protocols to be defined using *protocol descriptors* and *protocol diagrams*. These are explicitly linked to a set of scenarios, agents and messages. Rating: 7.

Selection Rules – Prometheus makes reference to “coverage, overlap and preferences”, but does not provide an explicit way of modelling these concepts. Rating: 2.

Prometheus provides quite good coverage of the key agent modelling concepts. Organisational modelling is not explicitly handled, but communication protocols are well defined.

6.8.3 Completeness and Clarity

Lifecycle coverage – Prometheus provides coverage from “system specification” through to “detailed design”. Its support tool [74] provides assistance with implementation in JACK [125] However, it does not include a dedicated requirements elicitation phase, and testing is not covered. Rating: 4.

Clear Models – Prometheus provides a detailed form-based means for defining models. However, these models are often general, and may vary in granularity. For example, the mechanisms for modelling data (beliefs) are not specified. Rating: 5.

Consistency – Prometheus includes a separate phase for consistency checking. The support tool also provides a degree of consistency checking support. Rating: 6.

Cooperation – Agent communication is well modelled by Prometheus, but organisational design is not covered. Rating: 5.

Concurrency – Agents in Prometheus are assumed to run concurrently, and interact via message passing. Detailed protocol diagrams are used. Rating: 6.
Iterative Support – In the definition of the Prometheus process [22], there is little reference to iteration. However, recent releases of the support tool do provide some support for iteration through code generation. Rating: 3.

Prometheus provides a reasonably clear mechanism for modelling multiagent systems. However, the process would benefit from taking a whole-of-lifecycle approach, modelling all phases of development from requirements elicitation through to testing and maintenance.

6.8.4 Usage and Practicality

Domain applicability – Prometheus is appropriate for building a range of agent systems. Its close links to the JACK implementation environment mean it is best used for projects intended for that platform. Prometheus is suited to creating closed agent systems, where relationships between agent types are well known. Rating: 5.

Scalability – Prometheus provides no guidelines for recommended project sizes, but due to its detail in specifying agent systems, the process should scale reasonably well. However, abstraction could be improved by the addition of concepts to enhance modularity, such as agent organisations. Rating: 5.

Abstraction – The Prometheus methodology allows for refinement to occur as the developer progresses through each phase of development. However, within each phase, organisational groupings could assist with abstraction. Rating: 6.

Usability – Prometheus provides a support tool to aid in usability, and the process has been well defined in a book dedicated to it. However, the implementation support is limited to a specific platform (JACK [125]), and the detailed design elements of Prometheus are largely oriented towards that platform. Rating: 6.

Learnability – The detailed description of the process and the provision of a support tool both aid in the learnability of Prometheus. However, the lack of
lifecycle coverage for requirements, and the reliance on outside modelling techniques for defining data may lead to confusion. Rating: 5.

*Reuse of models* – Prometheus provides capabilities for reuse via the definition of agent capabilities. Rating: 5.

*Heterogeneous agents* – Prometheus does not make reference to heterogeneous agent systems, and is closely coupled to JACK. Rating: 1.

Prometheus provides good coverage of the features for the “usage and practicality” category. Its only limitations are due to its close coupling to a specific implementation platform, and its lack of support for modularity within organisational design.

### 6.9 Comparison of Evaluation Results

The results of the evaluation may now be compared, by totalling the feature ratings for each process in each particular category. This will provide a high-level view of how well each process performs in each of the categories.

The results from the above evaluation are tabulated for the “Core Agent Properties” feature category in *Table 6.1*.

<table>
<thead>
<tr>
<th></th>
<th>NUMAP</th>
<th>Gaia</th>
<th>Prometheus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autonomy</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Reactiveness</td>
<td>7</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Proactiveness</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td><strong>Totals:</strong></td>
<td><strong>21</strong></td>
<td><strong>17</strong></td>
<td><strong>21</strong></td>
</tr>
</tbody>
</table>

*Table 6.1 – Core Agent Properties*
The evaluation results for the “Agent Modelling Concepts” feature category are presented in Table 6.2.

<table>
<thead>
<tr>
<th></th>
<th>NUMAP</th>
<th>Gaia</th>
<th>Prometheus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Belief</td>
<td>7</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Goal</td>
<td>7</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Plan</td>
<td>7</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>Organisation</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Protocol</td>
<td>7</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>Selection Rules</td>
<td>7</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td><strong>Totals:</strong></td>
<td><strong>47</strong></td>
<td><strong>22</strong></td>
<td><strong>38</strong></td>
</tr>
</tbody>
</table>

Table 6.2 – Agent Modelling Concepts

The “Completeness and Clarity” feature category has been assessed in Table 6.3.

<table>
<thead>
<tr>
<th></th>
<th>NUMAP</th>
<th>Gaia</th>
<th>Prometheus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lifecycle coverage</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Clear Models</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Consistency</td>
<td>5</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Cooperation</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Concurrency</td>
<td>6</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Iterative support</td>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><strong>Totals:</strong></td>
<td><strong>34</strong></td>
<td><strong>25</strong></td>
<td><strong>29</strong></td>
</tr>
</tbody>
</table>

Table 6.3 – Completeness and Clarity

The results for “Usage and Practicality” are presented in Table 6.4.

<table>
<thead>
<tr>
<th></th>
<th>NUMAP</th>
<th>Gaia</th>
<th>Prometheus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain applicability</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Scalability</td>
<td>6</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Abstraction</td>
<td>6</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Usability</td>
<td>6</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Learnability</td>
<td>6</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Reuse of models</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Heterogeneous agents</td>
<td>7</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td><strong>Totals:</strong></td>
<td><strong>42</strong></td>
<td><strong>23</strong></td>
<td><strong>33</strong></td>
</tr>
</tbody>
</table>

Table 6.4 – Usage and Practicality
In order to provide a clearer snapshot of how each process addresses each feature category, the total scores for each feature category can be tabulated, both as raw values and as a percentage.

<table>
<thead>
<tr>
<th>Feature Category</th>
<th>NUMAP</th>
<th>Gaia</th>
<th>Prometheus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Agent Properties</td>
<td>21</td>
<td>17</td>
<td>21</td>
</tr>
<tr>
<td>Agent Modelling Concepts</td>
<td>47</td>
<td>22</td>
<td>38</td>
</tr>
<tr>
<td>Completeness and Clarity</td>
<td>34</td>
<td>25</td>
<td>29</td>
</tr>
<tr>
<td>Usage and Practicality</td>
<td>42</td>
<td>23</td>
<td>33</td>
</tr>
<tr>
<td><strong>Totals:</strong></td>
<td><strong>144</strong></td>
<td><strong>87</strong></td>
<td><strong>121</strong></td>
</tr>
</tbody>
</table>

Table 6.5 – Feature Categories

<table>
<thead>
<tr>
<th>Feature Category</th>
<th>NUMAP</th>
<th>Gaia</th>
<th>Prometheus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Agent Properties (%)</td>
<td>100</td>
<td>81</td>
<td>100</td>
</tr>
<tr>
<td>Agent Modelling Concepts (%)</td>
<td>96</td>
<td>45</td>
<td>78</td>
</tr>
<tr>
<td>Completeness and Clarity (%)</td>
<td>81</td>
<td>60</td>
<td>69</td>
</tr>
<tr>
<td>Usage and Practicality (%)</td>
<td>86</td>
<td>47</td>
<td>67</td>
</tr>
<tr>
<td><strong>Totals (%):</strong></td>
<td><strong>89</strong></td>
<td><strong>54</strong></td>
<td><strong>75</strong></td>
</tr>
</tbody>
</table>

Table 6.6 – Feature Category Percentages

This information may then be presented in a graphical form, using a “radar” chart, to provide a visible indication of process coverage. See Fig. 6.1.

Fig 6.1 – Feature Category Radar Chart

218
The above analysis shows that both Prometheus and NUMAP are significantly more feature complete than the Gaia process. This is primarily due to their coverage of the detailed design process. Both processes are appropriate for use in real-world projects, where a feature-complete tool for designing multiagent systems is desired.

NUMAP scores higher overall, primarily due to its enhanced lifecycle coverage (including requirements elicitation), flexibility in supporting multiple agent implementation platforms, support for heterogeneous multiagent systems, an dependency tracking mechanism for iterative design.

In order to confirm this feature analysis, future work should involve a more comprehensive analysis by independent subjects. Further, the process should be subjected to real-world testing in large-scale projects, to demonstrate the benefits of its modularity, flexibility and real-world utility. This would also assess the scalability of the process, and provide vital feedback for future improvements.

6.10 Summary

This chapter presented a new evaluation framework for assessing the completeness of design processes for BDI multiagent systems. This work builds upon best practices in the literature, by identifying the features most widely assessed as being vital to multiagent design processes, and grouping them into intuitive categorisations.

The feature analysis was then used to assess the completeness of three processes intended for developing multiagent systems, including NUMAP. The results were tabulated and displayed graphically. The outcome of the evaluation showed that design processes must cover the full lifecycle of software development to be realistically considered “complete”.

219
Any such evaluation framework has a degree of subjectivity associated with the allocation of ratings to each feature. In order to alleviate this problem, justifications were presented for each rating, and existing evaluations of the processes were considered where feasible. In order to further address this problem, it would be advantageous to have multiple parties familiar with each of the design process independently conduct the evaluation.

This evaluation demonstrates that the NUMAP design process compares favourably with even the most well-established existing methodologies. Further refinements will be done in future to bring its feature-set even closer to the ideal embodied by the evaluation scheme.
Chapter 7 Conclusion

This chapter presents a summary of the work carried out, a summary of the contributions made, an account of the limitations of the work, and a discussion of directions for future research.

7.1 Chapter Synopsis

Chapter 1 provided an introduction the concept of multiagent systems, providing a brief description of what makes agents unique, and describing the concept of “Belief-Desire-Intention” (BDI) model of agent design.

The motivation of the work was presented – the desire to develop a process that uses the notions of “human action” to guide development from the earliest stages of requirements elicitation, right through to implementation. The focus of the research, on delivering practical deliverables, was described.

Chapter 2 provided necessary background information on the origins of multiagent systems, the development of the Belief-Desire-Intention model, and on existing processes that exist for developing multiagent systems.

Multiagent systems were defined, and their characteristics outlined. The need of a process for developing multiagent systems was established. Historical approaches to developing multiagent systems were presented, from the earliest approaches associated with robotics, through to the development of the BDI model, based on the philosophy of “human action”. Existing design approaches for developing multiagent systems were then presented.

A number of goals for the development of a new multiagent design process were then presented in detail.
Chapter 3 provided further detail on the advantages of multiagent systems, focusing specifically on BDI agents, and presented an overview of the tools available to construct them.

Chapter 4 introduced the Newcastle University Multi-Agent Process (NUMAP). The phases of the process were identified, and an overview of the process was presented.

Finally, a detailed specification of the process was presented. For each phase of the process, its purpose was highlighted. Any new concepts were introduced, and compared to those used by alternative design processes. The concepts used in each phase of the process were then specified in detail. Finally, the dependency tracking mechanisms used in NUMAP were described.

Chapter 5 provided a description of the NUMAP support tool. The goals for the support tool were introduced, its main features described, and the underlying design decisions discussed.

Chapter 6 described a feature-based evaluation framework for BDI multiagent systems. This feature analysis was used to assess the completeness of three processes for developing multiagent systems, including NUMAP.

7.2 Summary of Work Presented

In this work, the Newcastle University Multi-Agent Process (NUMAP) was presented. The motivation for NUMAP was to create a process that was grounded in the real-world practicalities of developing multiagent systems. A detailed and prescriptive process with clear definitions of process elements allows developers to easily follow the process and successfully create well-designed systems.

The focus of the process is primarily on developing BDI multiagent systems, although a modular design allows other system-development approaches to be used. The BDI model, inspired by the notions of human reasoning, provides a
natural progression in software abstraction. In order to take advantage of this higher level of software abstraction, specific design techniques are required. Using traditional approaches grounded in object oriented software development is a counterproductive approach, because they encourage the conceptualisation of the system to occur at a lower level, rather than allowing the entire process of software development to be oriented around BDI concepts.

The NUMAP process takes the notions of agents, goals and plans, and places them at the heart of every stage of the process. As developers progress from requirements specification, through to analysis and design, these concepts are gradually refined, adding additional detail, and introducing supplemental concepts, but the process, however, remains oriented around specifying agents, their goals, and how they go about meeting those goals.

In order to aid progression through the process, and to support iterative design, a dependency tracking mechanism was introduced. This mechanism allows dependencies to be tracked both within a phase and between phases. This is particularly vital for iterative development. For example, if an agile approach [46] is employed, the NUMAP process will be repeated many times, as new requirements are identified, and the software is incrementally refined. Allowing the impact of changes on individual elements of the design to be tracked greatly assists in seeing how changes are propagated throughout the design.

In order to support the developer in following the process, a support tool was developed. This support tool provides Java-based forms for the developer to use when specifying the details about each element of their design. “Guides” allow transitions between phases to be assisted, tracking dependency mappings between phases, and with data for subsequent phases automatically being filled where appropriate. Code template generation is supported for Jadex [39] and JACK [65], and an automatic document generation mechanism was implemented.
A new feature-based evaluation framework for BDI multiagent systems was detailed. This technique builds upon best practices in the literature, by identifying the features most widely assessed as being vital to multiagent design processes, and grouping them into intuitive categorisations. The feature analysis was then used to assess the completeness of three processes for developing multiagent systems, including NUMAP.

7.3 Contributions of this Work

The principal focus and contribution of this work is the Newcastle University Multi-Agent Process (NUMAP) for the development of BDI multiagent systems.

To develop this process, research was conducted into existing multiagent development processes, and the strengths and weaknesses of these processes were evaluated, highlighting to what extent they cover the software development lifecycle, how well they model agent concepts, and whether they allow software to be developed at an appropriate level of abstraction for BDI systems.

Agent runtime environments were then investigated, and considerable effort was put into gaining practical experience with developing multiagent systems. Based on this, the NUMAP was created, with the intent of designing a process that provides the most practical possible approach for developing multiagent systems.

The chief contribution of NUMAP is the creation of a process that is designed from start-to-finish to allow development of software by specifying the design in terms derived from the philosophical theories of “human action”. While other design processes for multiagent systems exist, none centre on the notions of agents, goals, beliefs and plans throughout the entire development process to the extent that NUMAP does. This approach provides an excellent fit with the BDI model of software agents.
As secondary contribution relates to the unique modular nature of NUMAP. While the focus of the work was on developing a process for creating BDI multiagent systems using a goal-oriented requirements phase, it was evident that the process could be used with a much wider variety of approaches. The requirements and analysis phases were defined in such a manner that an alternative requirements approach could be substituted, provided its outputs could be mapped to the format used in the analysis phase.

By creating specific “transitional mappings” between phases, the requirements phase became a standalone module that could be replaced with an alternative approach, provided mappings were also created for that new module. The separation of phase transitions into a separate, standalone task led to a modular design, in which each phase of the process was independent.

This modularity also allows for alternative agent design approaches to be used seamlessly. A development project could conceivably include a combination of agents designed with a BDI approach, and other agents designed with an alternative approach, such as object-oriented software wrapped in a software layer that supports agent services. Both techniques could be supported within NUMAP, simply by defining a new agent module for the object-oriented approach.

A comprehensive dependency tracking mechanism has also been proposed, to assist with iteration through the process, and modification of the design. Within each phase of the process, changes to a modelled entity (e.g. a goal) will affect all other entities that rely upon it (e.g. agents using that goal). By specifying how these relationships are mapped, the effects of a change can be readily tracked.

However, the dependency tracking mechanism goes further than this. By taking advantage of the mappings used when transitioning between phases, the technique can be extended to track dependencies between phases of the process.
Changes to an element of the design in the one phase will affect all elements that are derived from that in subsequent phases. For example, a goal may be specified at the requirements phase, and then refined into multiple goals at the analysis phase. Each of those derived goals will be affected by any change to the goal in the requirements phase. The inter-phase dependency tracking mechanism allows for these effects to be tracked throughout the process.

To fully take advantage of the advantages of NUMAP, a software support tool was required. This support tool assists the developer in following the process, allowing them to specify all required data for each phase. It also guides the developer through each stage of the process, tracking the mappings between phases, to allow for reporting of dependencies. Finally, it allows code templates to be generated for both Jadex [39] and JACK [65], these being pre-eminent implementation environments.

A new, comprehensive feature-based evaluation framework for BDI multiagent systems was been presented. To develop this framework, existing frameworks that use feature-analysis for evaluating multiagent systems were investigated. The features used by these techniques were assessed, and the most commonly selected criteria were identified. The selection of features struck a balance between selecting the most common features, avoiding overlap of similar properties, and avoiding gaps in the set of features to be evaluated.

The feature analysis was then used to assess the completeness and effectiveness of three multiagent development processes, including NUMAP. The information was presented in both a tabular and graphical form, and the limitations of each process were highlighted.
7.3.1 Primary Contributions

The primary contributions from this work are as follows:

- The Newcastle University Multi-Agent Process (NUMAP). A new process for the development of BDI multiagent systems, including full specifications of all phases of the process and the mappings between them.

- The modular design approach used by NUMAP, including a strict division between each phase of the process, with well-defined transitions between each phase.

- The dependency tracking mechanism used in the process, including a way of tracking the effects of changes both within phases and between them.

- The NUMAP support tool, which assists the user in completing each phase of the process, transitioning between phases, tracking dependencies, generating documentation, and generating code templates.

- A new feature-based analysis method for evaluating design processes for BDI multiagent systems.

7.4 Limitations of the Work

While NUMAP provides a comprehensive, usable approach for developing software using multiagent systems, it does have some limitations:

- Organisations within NUMAP are strictly a concept for componentising the system to manage complexity. Advanced concepts such as dynamic reconfiguration of organisations, admission criteria, trust modelling, rules and norms have not been addressed.

- NUMAP does not directly address testing and maintenance. While the dependency tracking mechanism may help in this regard, tools and
techniques for these phases of development have not been specifically developed.

- The NUMAP support tool does not perform consistency checking of the models created.
- Comprehensive user testing of the process has not yet been performed, to assess its usability by a range of developers.

### 7.5 Future Work

This work provides a comprehensive process for developing multiagent systems, but more can be done to extend it to cover additional areas, and address the limitations listed in the previous section.

NUMAP would benefit from a revised “organisational design” phase to provide support for dynamic organisation of agents. This was not covered in this work, as it is generally not supported at a software level in existing agent implementation platforms. Managing organisations at runtime is a complex research problem in its own right, generally involving software extensions to the agent platform [165, 166]. Investigation of such software extensions could be conducted, and the organisational design process amended to support optional concepts that facilitate the implementation of dynamic organisations.

Investigating integration of NUMAP’s concepts with the Agent OPEN Process Framework [75] would also be a possibility for the future. While the focus of this work has been on creating a cohesive, integrated approach to multiagent systems development, Agent OPEN’s method-engineering technique presents an interesting alternative. It may be possible to segment NUMAP into process fragments and integrate these with the Agent OPEN repository.

Testing and maintenance have not been covered within NUMAP. However, for the process to be truly considered to cover the “full lifecycle” of development, these stages of development should be covered. Taking advantage of existing techniques for unit testing of multiagent systems [167] would assist in this
regard. However, to fully support the developer in the testing process, tool support will be required, which will necessarily be tied to a particular implementation platform. Dependency tracking features do, however, assist in assessing how changes to code will be propagated through the software.

The NUMAP support tool should also be extended to support other additional features, such as consistency checking. Improvements to its user interface could also improve its usability. At present, the emphasis has been on developing a functional piece of software, rather than an attractive one.

Finally, a comprehensive user evaluation of NUMAP would be desirable, to evaluate the usability of the process among those unfamiliar with it, and in particular, among developers unfamiliar with multiagent systems development. In particular, this should include a feature-based analysis by independent subjects.
Appendix A – Classifying Agents

A number of classification schemes, or typologies for describing agents were outlined in Chapter 2. Based on these existing classification schemes, a new, more comprehensive classification scheme can be created. The core attributes of software agents are autonomy, temporal continuity and social ability. Agents must be able to run independently, with little or no human intervention, therefore autonomy is a necessary property of agency. Temporal continuity is required, as agents must execute continuously, rather than simply perform a task, and terminate.

Agents must also possess some form of social ability. The real advantages of software agents come not from individual pieces of software acting in isolation, but from communities of interacting agents.

In addition to the core attributes, agents may be classified according to the following features:

- Pro-activeness
- Adaptiveness
- Mobility
- Collaboration
- Veracity
- Disposition

Each of these features may be further sub-divided into a list of properties, as explained below.

**Pro-activeness**

An agent’s pro-activeness refers to how it reacts to - and reasons about - its environment, and how it pursues its goals. Given that the purpose of an agent is to autonomously and continuously perform a given set of tasks on behalf of a
requester, the approach that the agent takes toward achieving goals is central to its performance. An agent's pro-activeness may be characterised as one of the following:

**PURE REACTION:** Pure Reaction is the simplest form of behavior, and involves directly reacting to stimuli in the agent's environment, by mapping an input from their sensors directly to an action. This approach has two principal advantages, namely, the agent can react quickly to external events, and it greatly simplifies the process of designing agents. Despite the apparent simplicity of reactive agents, complex behaviors can evolve from interaction with complex environments. Brooks [56] devised a reactive framework called the subsumption architecture for physical robots, which uses layers of reactive control systems to achieve complex behavior.

**PURE PLANNING:** Pure planning lies at the other end of the spectrum, and involves agents taking a purely planning, or goal-oriented, approach to achieve their goals. This approach relies upon utilising planning techniques from traditional AI to identify tasks that need to be performed in order to satisfy the goals of the agent. This approach allows flexibility in the pursuit of goals, but is often slow. The dominant technique for goal-directed agent behavior is the “Belief, Desire, Intention” (BDI) model [168].

**HYBRID:** Hybrid agents combine the above two techniques, incorporating both reactive and planning components. This approach merges the rapid response of reactive agents with the sophisticated reasoning of planning agents, and is therefore widely viewed as the superior approach to agent design. In fact, the notion of agency presented by Wooldridge and Jennings [169] identifies both reactivity and pro-activeness as necessary attributes of all agents. Parunak [170] divides hybrid agents into two classes: “reaction overridden by plan”, where the planner may overrule the reactive component if it disagrees with it, and “reaction modified by plan”, where the planner can reconfigure the reactive component to behave differently in the future. The latter technique requires a degree of adaptiveness within the agent.
Adaptiveness

Adaptiveness describes an agent’s ability to modify its behavior over time. This is a key attribute that is often associated with agents. In fact, the term “agent” is often taken to implicitly mean “intelligent agents”, which combine traditional AI techniques to assist in the process of autonomously performing tasks.

Adaptiveness is closely related to pro-activeness, with many pure planning or hybrid systems relying on the ability to adapt. Despite this, not all agents are adaptive, and some only adapt in a limited manner. Adaptiveness may fall into several different categories as noted below:

**LEARNING**: Learning agents have the capacity to modify their behavior over time in order to adapt their functionality to their environment, and to improve their effectiveness. A wide range of techniques have been applied to learning agents, including memory-based learning, reinforcement learning [171], and Bayesian belief networks [172].

**SUBSUMPTION**: The Subsumption Architecture allows the designer to add additional “layers of competence” to an agent over time. It was first defined by Brooks [56] as an architecture for autonomous robots, but has since been adapted for software agents [57]. The agent designer can hence expand and adapt the agent’s functionality over successive iterations of development. This differs from agent learning, as the adaptation is performed by designers explicitly adding functionality to the agent.

**NON-ADAPTIVE**: Non-adaptive agents are those that do not modify their behavior over time. As noted by Wooldridge [173], although the discipline of “intelligent agents” largely grew out of the field of AI, not all agents need to be capable of learning. The only intelligence that is required by agents is the capability to make independent decisions, i.e. to act autonomously. While learning is often an appropriate technique for agents to employ, its usefulness will depend on the circumstances in which the agent is being used. In mission critical applications,
for example, adaptiveness may be a liability, as it could lead to unpredictable behavior by the system.

**CONSTRAINT BASED:** Constraint-based agents place restrictions on the agent’s capacity to adapt, and so are able to mitigate the problems associated with learning agents in critical systems. This allows many of the benefits of learning agents, while providing safeguards to ensure that the agent still fulfills critical functions.

**Mobility**

Software agents are well-suited to tasks involving large-distributed networks such as the Internet. Consequently, much of the research into agents has revolved around the concept of mobility. An agent may be:

**PHYSICALLY MOBILE:** Physically mobile agents are capable of transporting their execution between machines on a network. This provides an attractive mechanism for developing software for distributed systems. Mobility is often implemented in a transparent manner, allowing the agent to continue normal execution as it travels around the network. An example of this approach is the Concordia agent [174], developed by Mitsubishi Electric ITA, in which migration between network nodes is transparent to the agent.

**LOGICALLY MOBILE:** Logically mobile agents are those that physically execute on a single machine, but access other logical locations, via a network connection. These agents may be thought of as visiting these locations in a conceptual sense, although their actual execution is fixed to a single physical machine. Logically mobile agents provide a suitable mechanism for gathering data from the Internet. A typical example of such an agent is a web spider [175], which visits and processes a series of web pages by following hypertext links, for the purpose of cataloguing websites.

**STATIC:** Static agents are those do not provide a mechanism for mobility.
Collaboration

Collaboration among agents underpins the success of an operation or action in a timely manner. For this reason, agents should possess some form of social ability, and this may be divided into two types:

COMMUNICATIVE: Communicative agents are those that are able to coordinate with other agents by sending and receiving messages using some form of agent communication language. This allows a high degree of collaboration, with social activities such as distributed problem solving and negotiation being possible. Several agent communication languages are available, the most prominent being KQML [176].

NON-COMMUNICATIVE: Non-communicative agents are those that do not engage in formal communication. Although direct agent communication is desirable in many situations, it is possible for agents to collaborate without actual communication taking place. Interaction of agents with resources and their environment may lead to collaborative or competitive behavior emerging. For example, the SWARM [60] agent system provides a framework within which communities of agents can interact. The SWARM system is generally used for simulating social systems, and many such simulations demonstrate collaborative behavior without direct communication. A simple example of this is the HeatBugs [61] model.

Veracity

Collaborating agents, whether communicative or non-communicative, may attempt to deceive other agents via their messages or behavior. Agents may hence be classified by their veracity:

TRUTHFUL: Truthful agents are those that do not attempt to intentionally deceive other agents. In a closed environment, where the veracity of all agents is guaranteed, negotiation and interaction is greatly simplified. If an agent
indicates that it can provide a service, other agents can assume that it will make an attempt to provide it. If an agent provides information to assist in satisfying a goal, all other agents can be reasonably certain that this information is correct.

**UNTRUTHFUL:** Untruthful agents are those that may attempt to deceive other agents either through the provision of false information, or by acting in a deceptive manner. In an open environment, where various agents from different vendors compete to achieve their own goals, the issue of deception becomes a factor. In such a system, collaboration becomes much more difficult.

**Disposition**

The final factor in this classification of agents indicates the “attitude” of the agent toward other agents, and its willingness to cooperate with them. Agents may be classified as:

**BENEVOLENT:** Benevolent agents are those that always attempt to perform a task when it is requested of them. Wooldridge [173] identifies benevolence as “the assumption that agents share common goals, and that every agent will therefore do what is asked of it”. As with truthfull agents, collaboration is greatly simplified in a system that consists entirely of benevolent agents.

**SELF-INTERESTED:** Self-interested agents are those that act in their own interest, collaborating with other agents only when it is beneficial to do so. Unlike benevolent agents, they cannot be expected to do what is asked of them, so coordinating with such agents becomes a difficult task. In many cases, self-interested agents may be competitive, vying with other agents to achieve a given goal, or acting to secure a better deal than other agents. A typical example of this would be an electronic auction system.

**MALEVOLENT:** Malevolent agents are those that attempt to inconvenience other agents, or undermine them in some way. Unlike self-interested agents, they do not simply act to achieve their own goals, they act in some predefined malicious
manner. While the difference between self-interested agents and malevolent ones may simply be a distinction in the goals of the agent, self-interested agents may have a positive impact on the system, while the presence of malevolent agents within a system is unlikely to be of any benefit. Some worms may be seen as a type of malevolent agent, as they execute autonomously, without user intervention, but are a malign influence on the system.

**Classification Summary**

Overall, all software agents can be seen as software components with *autonomy, temporal continuity and social ability*. Additionally, agents have varying degrees of reactiveness, pro-activeness, mobility, collaboration, and disposition, and can be classified according to these properties.
Appendix B - Mappings Between Phases

In Chapter 4, the notion of mapping concepts between phases was introduced. At the end of each phase, the information collected in that phase can be used to populate initial values for the next phase.

This process is aided by a “wizard” in the NUMAP support tool, which supports advanced features, such as refining an element (e.g. a goal) from one phase of the process into multiple elements in the next phase (e.g. decomposing a requirements goal into multiple goals at the analysis phase).

These mappings play an important part in NUMAP's inter-phase dependency tracking system.

The details of the mappings between phases are presented in the remainder of this appendix.
Mapping Requirements Phase to Analysis Phase

The mappings between the Requirements Phase and the Analysis Phase are presented in the following table.

Please note, these mappings are specific to the GBRAM Requirements module. If another module were used, a different mapping scheme would be needed.

<table>
<thead>
<tr>
<th>GBRAM Agent</th>
<th>Analysis Agent Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Type</td>
<td>Type</td>
</tr>
<tr>
<td>Description</td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td>Other fields are left blank</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GBRAM Goal</th>
<th>Analysis Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Type</td>
<td>Type</td>
</tr>
<tr>
<td>Description</td>
<td>Description</td>
</tr>
<tr>
<td>Actions</td>
<td>Plans (Matched by name)</td>
</tr>
<tr>
<td>Obstacles</td>
<td>Obstacles</td>
</tr>
<tr>
<td>Scenarios</td>
<td>Scenarios</td>
</tr>
<tr>
<td>Preconditions</td>
<td>Preconditions</td>
</tr>
<tr>
<td>Postconditions</td>
<td>Postconditions</td>
</tr>
<tr>
<td>Parent</td>
<td>Parent</td>
</tr>
<tr>
<td></td>
<td>Other fields are left blank</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GBRAM Action</th>
<th>Analysis Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Description</td>
<td>Description</td>
</tr>
</tbody>
</table>
|              | Other fields do not have a direct correlation, but may be used to assist in identifying analysis properties.

Table B.1 – Requirements – Analysis Mappings

NOTE: In contrast to GBRAM, in the analysis phase, *goals relevant to an agent are listed in the agent specification* (rather than agents being listed as part of goals). During the transition between the Requirements Phase and the Analysis phase, these relationships between goals and agents *will need to be transposed.*
**Mapping Analysis Phase to Design Phases**

At the completion of the Analysis phase, concepts are mapped to both the Organisational Design phase and the Detailed Design phase. The following table shows these mappings.

Mappings to the Organised Design phase are prefixed with an “O”. Mappings to the Agent (or Detailed) Design phase are prefixed with a “D”.

Please note, the mappings to the Agent Design phase are for the default NUMAP BDI Agent Design module. If a different module was to be used in this phase, to support an alternative agent design approach, the mappings would need to be revised.

<table>
<thead>
<tr>
<th>Environment Element</th>
<th>O: Environment Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Description</td>
<td>Description</td>
</tr>
<tr>
<td>Attributes</td>
<td>Attributes</td>
</tr>
<tr>
<td>Parent</td>
<td>Parent</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Agent Type</th>
<th>O: Agent Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Description</td>
<td>Description</td>
</tr>
<tr>
<td>Roles</td>
<td>Roles</td>
</tr>
<tr>
<td>Goals</td>
<td>Goals</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Agent Type</th>
<th>D: Agent Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Description</td>
<td>Description</td>
</tr>
<tr>
<td>Sensors</td>
<td>Sensors</td>
</tr>
<tr>
<td>Effectors</td>
<td>Effectors</td>
</tr>
<tr>
<td>Plans</td>
<td>Plans</td>
</tr>
<tr>
<td>Goals</td>
<td>Goals</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Goal</th>
<th>O: High-Level Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Type</td>
<td>Type</td>
</tr>
<tr>
<td>Description</td>
<td>Description</td>
</tr>
<tr>
<td>Parent</td>
<td>Parent</td>
</tr>
</tbody>
</table>

(continued on next page)
<table>
<thead>
<tr>
<th>Goal</th>
<th>D: Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Type</td>
<td>Type</td>
</tr>
<tr>
<td>Description</td>
<td>Description</td>
</tr>
<tr>
<td>Scenarios</td>
<td>Failure Condition</td>
</tr>
<tr>
<td>Constraints</td>
<td>-</td>
</tr>
<tr>
<td>Preconditions</td>
<td>Preconditions</td>
</tr>
<tr>
<td>Postconditions</td>
<td>Postconditions</td>
</tr>
<tr>
<td>Parent</td>
<td>Parent</td>
</tr>
<tr>
<td>Note: Constraints should be refined into postconditions by this point.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Organisation</th>
<th>O: Organisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Description</td>
<td>Description</td>
</tr>
<tr>
<td>Interaction Organisations</td>
<td>Interaction Organisations</td>
</tr>
<tr>
<td>Member Agents</td>
<td>Member Agents</td>
</tr>
<tr>
<td>Parent</td>
<td>Parent</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Role</th>
<th>O: Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Description</td>
<td>Description</td>
</tr>
<tr>
<td>Provided Services</td>
<td>Provided Services</td>
</tr>
<tr>
<td>Used Services</td>
<td>Used Services</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sensor</th>
<th>D: Sensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Description</td>
<td>Description</td>
</tr>
<tr>
<td>Environment Element</td>
<td>Environment Element</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Effector</th>
<th>D: Effector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Description</td>
<td>Description</td>
</tr>
<tr>
<td>Environment Element</td>
<td>Environment Element</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Service</th>
<th>O: Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Description</td>
<td>Description</td>
</tr>
<tr>
<td>Other fields are left blank</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Plan</th>
<th>D: Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Description</td>
<td>Description</td>
</tr>
<tr>
<td>Sensors Used</td>
<td>Sensors Used</td>
</tr>
<tr>
<td>Effectors Used</td>
<td>Effectors Used</td>
</tr>
<tr>
<td>Services Used</td>
<td>Services Used</td>
</tr>
<tr>
<td>Satisfied Goals</td>
<td>Satisfied Goals</td>
</tr>
<tr>
<td>Other fields are left blank</td>
<td></td>
</tr>
</tbody>
</table>

**Table B.2 – Analysis – Design Mappings**
Mapping Organisational Design Phase to Agent Design Phase

At the conclusion of the Analysis phase, a variety of concepts will have been mapped directly to their counterparts the Agent Design phase.

However, during refinement in the Organisational Design phase, new agent types may have been introduced during the process of refinement of agents. The internal details of these agents will be specified in Detailed Design, and mappings should be created between them.

At the commencement of the Agent Design phase, the NUMAP support tool handles this in the following manner:

• Any agent type data from the analysis phase that has been automatically populated is left as-is. This preserves the sensor, effector, plan and goal information.
• Any new agent types that have been created during the organisational design phase are automatically created in the detailed design phase. However, they are created as an “empty shell”, with only their name, description and parent information populated. This is necessary, as the other concepts are not specified in any detail within organisational phase agent types.
• Mappings are updated, so the detailed design agent types are linked to their organisational design counterparts.

This preserves the concepts identified during analysis, while ensuring a proper mapping between Organizational Design and Agent Design phases.
References


Reasoning System," in Proceedings of the 11th International Joint

Systems: A Case Study," IEEE TRANSACTIONS ON SYSTEMS, MAN, AND

Paper.


[71] FIPA. (2003, 2013). The FIPA Agent UML Modelling Web Site,

[72] M.-P. Huget, "Agent uml notation for multiagent system design," Internet


[74] RMIT Intelligent Agents Group. (2013). Prometheus Design Tool,

OPEN with concepts used in the Tropos methodology," in ESAW03,

[76] B. Henderson-Sellers, "Creating a Comprehensive Agent-Oriented
Methodology," in Agent-Oriented Methodologies, B. Henderson-Sellers
368-386.

Available: http://www.fipa.org/specs/fipa00029

[78] X. Mao and E. Yu, "Organizational and Social Concepts in Agent Oriented

[79] J. J. Odell, H. V. D. Parunak, and M. Fleischer, "The role of roles in
designing effective agent organizations," in Software engineering for


251


